Algorithms and Methods for High-Performance Model Predictive Control

Frison, Gianluca

Publication date:
2016

Document Version
Publisher's PDF, also known as Version of record

Algorithms and Methods for High-Performance Model Predictive Control

Gianluca Frison

Kongens Lyngby 2015

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

PhD-2015-402
ISSN-0909-3192

Summary (English)

The goal of this thesis is to investigate algorithms and methods to reduce the solution time of solvers for Model Predictive Control (MPC). The thesis is accompanied with an open-source toolbox for High-Performance implementation of solvers for MPC (HPMPC), that contains the source code of all routines employed in the numerical tests. The main focus of this thesis is on linear MPC problems.

In this thesis, both the algorithms and their implementation are equally important. About the implementation, a novel implementation strategy for the dense linear algebra routines in embedded optimization is proposed, aiming at improving the computational performance in case of small matrices. About the algorithms, they are built on top of the proposed linear algebra, and they are tailored to exploit the high-level structure of the MPC problems, with special care on reducing the computational complexity.

Summary (Danish)

Målet med denne afhandling er at undersøge algoritmer og metoder til at reducere beregningstiden for Model Prædiktiv Kontrol (MPC). Afhandlingen indeholder også en open-source værktøjskasse for High-Performance implementering af løsere for MPC (HPMPC). Værktøjskassen indeholder kildekoden til algoritmer og numeriske tests. Hovedfokus af afhandlingen er på lineær MPC.

I denne afhandling er algoritmerne og deres implementering lige vigtige. Implementeringsmæssigt, udforskes en implementationsstrategi der benytter kompakt lineær algebra-rutiner i den integrerede optimering. Denne metode forbedrer ydeevnen for mindre størrelse problemer. Algoritmemæssigt, baseres disse på den før omtalte kompakte lineære algebra. Algoritmerne skræddersyes også til at udnytte strukturen i MPC problemerne med specielt fokus på at reducere beregningskompleksiteten.

Preface

This thesis was prepared at the Department of Applied Mathematics and Computer Science (DTU Compute, formerly known as DTU Informatics) at the Technical University of Denmark, in partial fulfillment of the requirements for acquiring a PhD degree in Engineering. The PhD project was founded entirely by DTU Compute; three months as Research Assistant at DTU Compute were founded by EUDP 64013-0558 "Energy Efficient Process Control", and The Danish Council for Strategic Research in the project "CITIES - Centre for IT-Intelligent Energy Systems in Cities" (1305-00027B). All are gratefully acknowledged.

The thesis deals with algorithms and methods for the implementation of fast solvers for model predictive control. The focus of the thesis is on both the optimization algorithms (tailored to exploit the special structure of the model predictive control problem) and the implementation (thanks to a novel implementation strategy for the dense linear algebra routines in embedded optimization). All solvers and routines are gathered in the open-source toolbox HPMPC.

The thesis is in the form of a monograph. The main reason for opting for a monograph instead of a collection of papers is the desire to present the matter in many more details and in a more systematic way than possible in a paper. In particular, the first part of the thesis proposes a novel implementation strategy for the linear algebra routines in embedded optimization, and deep insight and extensive numerical tests are necessary to convince the reader of the effectiveness of the approach.

Kgs. Lyngby, 30-December-2015

Gianluca Frison

Acknowledgements

First and foremost, I would like to thank my supervisors. John Bagterp Jørgensen for allowing me to pursue what interested me the most, and for his precious advice. Niels Kjølstad Poulsen for his presence and support throughout the project.

I would also like to thank Moritz Diehl for opening me the doors of IMTEK, both during a research stay there, and now for a new adventure.

I would like to thank Milan Vukov and D. Kwame Minde Kufoalor (aka Giorgio) for the endless hours of enthusiastic work together. This project owes you a lot.

A big acknowledgement goes also to the defence opponents, Allan Peter Engsig-Karup, Daniel Axehill, Hans Joachim Ferreau, for their challenging questions and valuable feedback on the thesis.

Finally, I would like to thank my girlfriend Marie, my family and my friends for their love and their constant support in the good as well as in the difficult moments. I am glad I lived this adventure with you.

List of abbreviations

ADMM Alternating Direction Method of Multipliers

AS Active-Set

CPU Central Processing Unit

flop floating-point operation

FMA Fused Multiply-Add

FP Floating-Point

Gflops Billions of floating-point operations per second

IPM Interior-Point Method

ISA Instruction Set Architecture

LLC Last Level of Cache

LP Linear Programming

memop memory operation

MHE Moving Horizon Estimation

MMU Memory Management Unit

MPC Model Predictive Control

NMHE Nonlinear Moving Horizon Estimation

NMPC Nonlinear Model Predictive Control

OCP Optimal Control Problem

QP Quadratic Programming

SIMD Single-Instruction Multiple-Data

SQP Sequential Quadratic Programming

TLB Translation Lookaside Buffer

Contents

Chapter 1

Introduction

The aim of this thesis is to investigate algorithms and methods to reduce the solution time of solvers for Model Predictive Control (MPC). The thesis is accompanied with an open-source toolbox for High-Performance implementation of solvers for MPC (HPMPC), that contains the source code of all routines employed in the numerical tests [30]. The main focus of this thesis is on linear MPC problems, in that they arise as sub-problems in some Nonlinear MPC (NMPC) formulations.

1.1 Background

MPC is an advanced control technique that gained much attention in both academia and industry in the last few decades, for both the linear and nonlinear MPC cases [69]. Good introductions to MPC can be found in [61, 72, 71]. In a few sentences, MPC makes use of a model of the controlled plant to predict its future state and compute an input (also known as controls or manipulated variables) sequence optimal with respect to both performance metrics and plant constraints. MPC can deal with complex plants with several inputs and outputs (also known as controlled variables) and relative constraints, and incorporates in a predictive way information about set-point changes or known disturbances. This is achieved by solving at each sampling instant an optimization problem,

that is parametrized with respect to the current state of the plant. The need to efficiently and reliably solve this optimization problem at each sampling instant, as soon as a new estimate of the plant state is available, has traditionally limited the application of MPC to the control of plant with slow dynamic. There has been much research effort in improving the solution time of the optimization problems in MPC, that is a necessary condition for the applicability of MPC to the control of systems characterized by faster dynamic.

In the last few decades, MPC has been successfully applied to the control of systems characterized by increasingly faster dynamic, from the sampling times in the order of minutes all the way down to sampling times up to MHz rates [49]. Much of these improvements have been achieved thanks to the development of optimization algorithm tailored to the special structure of the MPC problem. The two main research directions in this field are off-line and on-line solution methods of the MPC problem.

Explicit MPC [21, 20] exploits the fact that the solution of the constrained linear MPC problem is a continuous piecewise affine function of the state over a polyhedral partition, and that it can be precomputed off-line for all initial states. The on-line part of the algorithm reduces to a lookup table, and therefore extremely high control frequencies can be achieved. Since in the worst case the number of regions can be as large as the combinations of active constraints, the use of explicit MPC is generally limited to very small MPC problems.

Alternatively, the solution of the optimization problems can be computed on-line, between two sampling instants. This imposes tight real-time requirements on the execution time of the solvers, requiring the development of reliable solution methods for optimization problems with the structure of MPC problems. Suitable solvers must be certified to return the solution within the available time, or at least should return a reasonable approximation of the solution if stopped early. On-line methods comprise first and second order optimization methods.

First order optimization methods (such as gradient methods [73, 50, 58] and splitting methods [82, 68]) are generally straightforward to implement and can easily exploit sparsity pattern and special structure of problems. They perform many but cheap iterations, where the cost-per-iteration is quadratic in the input and state size (requiring level 2 BLAS operations as e.g a matrix-vector multiplication or solution of a system of linear equation whose matrix is factorized off-line). In general, the number of iterations (and therefore the solution time) can vary significantly with the number of active constraints and the problem conditioning. Therefore, there has been much effort in finding certification on the solution time of these methods [74].

Second order optimization methods make use of second order information to converge to a solution in fewer iterations. Interior-Point Methods (IPM) and Active-Set methods (AS) belong to this class. In the IPM case, there exists a polynomial certification on the number of iterations required for convergence, but it is very loose and therefore of no practical value. In practice, in IPMs the number of iterations is rather unaffected by the problem instance: if well initialized, these methods can typically converge to the solution in 8-15 iterations. This justifies the wide use of IPMs in MPC [70, 89, 62, 26]. Each iteration is more computationally heavy than in the first order methods case, requiring the factorization and solution of a system of linear equations in the computation of the search direction. The factorization makes use of level 3 BLAS and therefore requires a cubic number of flops in the input and state size.

There is not a polynomial certification for the AS methods, that in the worst case can require an exponential number of iterations to converge. However, in practice AS methods perform well, and can return a solution quickly, and therefore they find wide applicability in MPC [54, 66]. Particularly interesting is the approach employed in the open-source AS solver qpOASES [28, 29], that, in case of early stop, returns the solution of a QP that is between the one solved at the previous sampling instant and the current one. AS methods typically require more iterations than IPM to converge, but the iterations are generally cheaper, since the KKT matrix can be updated without the need to re-factorize it, when there are changes in the active set.

The other factor contributing to the reduction in solution times has been the improvement in the computer hardware. In particular, processor frequencies have increased exponentially for about two decades, going from a few MHz at the beginning of the '80s, to about 3 GHz with the introduction of Intel Pentium 4 in 2002. In general, an increase in the CPU frequency translates directly in a similar increase in performance. Therefore, there has not been the need to invest much research effort on the implementation side. However, since then the CPU frequencies have stalled, and further improvements in computing power can come only from the use of vector execution units or multiple CPU cores, both requiring additional programming effort [84].

Probably due to the fact that matrices in MPC problems are generally of small size, the use of optimized BLAS libraries has not been considered of much help in implementing fast solvers for MPC or MHE [46]. An implementation technique that recently gained much traction in the field of MPC is code generation, originally proposed in this field in [62]. The idea of code generation is to exploit the fact that, in MPC, problems of fixed structure are repeatedly solved at each sampling instant. This can be done e.g. by choosing the best solution strategy for the problem at hand, or by exploiting sparsity and knowledge about the size of matrices. In the implementation of linear algebra routines, all loops can

be totally unrolled (as in [62]), or the size of the loops can be fixed at code generation time (as in [26]).

1.2 Thesis approach

As stated in the title, the thesis deals with algorithms and methods for fast model predictive control. Methods mainly refers to Part I of the thesis (dealing with linear algebra implementation methods), while algorithms mainly refers to Part II and Part III of the thesis (dealing with tailored algorithm for MPC). Both algorithms and their implementation are considered equally important in the development of fast solvers.

The structure of the thesis follows the structure of the HPMPC toolbox, as shown on the right hand side of Figure 1.1. Part I deals with the efficient implementation of linear algebra routines for embedded optimization, and that forms the basis for the developed solvers. Part II deals with solvers for unconstrained MPC and MHE problems. Part III deals with solvers for linear (constrained) MPC and MHE problems.

The approach used in the solvers development is to implement a toolbox of efficient dense linear algebra routines, and to explicitly exploit the structure and sparsity pattern of the MPC problems in the algorithms implemented using these routines.

Part I of the thesis proposes a novel implementation strategy for dense linear algebra routines, specially tailored for embedded optimization. The matrices of interest in embedded optimization are typically of small to medium size, and they can generally fit in cache. Therefore, some of the implementation techniques employed in optimized BLAS libraries (such as blocking for cache) give no advantages, on the contrary they decrease the performance due to the overhead of performing useless operations (that is particularly true in case of small matrices). Therefore, a subset of BLAS and LAPACK has been re-implemented using only techniques beneficial to the embedded optimization case, such as blocking for registers and explicit use of vectorization through SIMD instructions. Only single-thread code has been considered in this thesis, as parallel computation is mostly beneficial for larger size problems, and in any case it should be employed only once the single thread performance has been well optimized.

Furthermore, a special matrix format is proposed, called panel-major matrix format. It consists of horizontal panels (i.e. sub-matrices with few rows and many columns) of fixed height b_s stored one after the other, with the elements

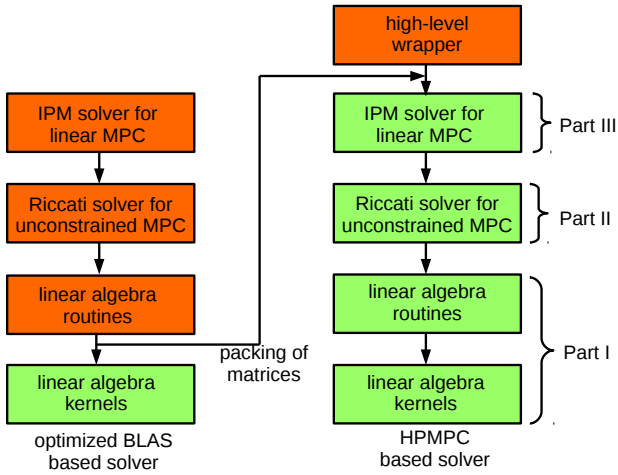


Figure 1.1: Structure of a Riccati-based IPM for linear MPC problems when implemented using linear algebra in either optimized BLAS or HPMPC. Routines in the orange boxes use matrices in column-major format, routines in the green boxes use matrices in panel-major format (or equivalent internal format in optimized BLAS). The thesis follows the structure of the HPMPC toolbox, with Part I dealing with linear algebra, Part II with solvers for unconstrained MPC problems and Part III with solvers for linear (constrained) MPC problems.

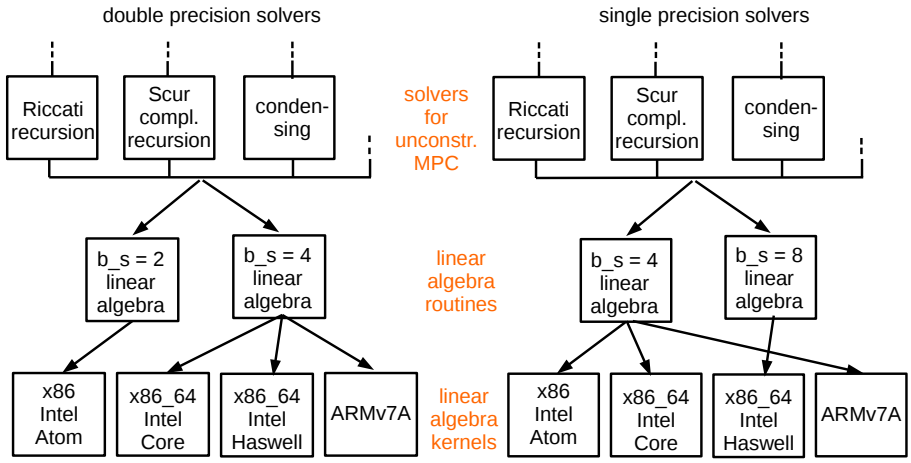


Figure 1.2: Structure of the linear algebra routines in HPMPC. The linear algebra kernels are tailored to each computer architecture. The linear algebra routines depend only on the panel height b_s (that may be different for single and double precision). The routines at higher levels in the routines hierarchy are completely architecture-independent.

within each panel stored in column-major order. This matrix format roughly corresponds to the innermost level of packing employed in optimized BLAS libraries, giving optimal performance for matrices fitting in cache (as typical in embedded optimization). The fundamental difference is that in optimized BLAS libraries the packing of matrices into this format is done at each call to a linear algebra routine, while in the proposed approach the panel-major matrix format is the format expected by linear algebra routines, as shown in Figure 1.1. This moves the overhead of packing matrices from the standard column-major or row-major formats much higher in the routines hierarchy. In particular, in embedded optimization the packing overhead can be well amortized over the iterations of the optimization methods.

The innermost loop of each linear algebra routine is coded in a separate function, the kernel. The linear algebra kernels are typically coded in assembly or using intrinsics, and are tailored for a number of computer architectures, as shown in Figure 1.2. The two outermost loops of linear algebra routines are almost architecture-independent, in the sense that they are coded in C code but they depend on the panel height b_s. All routines at higher levels in the routines hierarchy are completely architecture-independent.

Part II of the thesis deals with algorithms for unconstrained MPC and MHE problems. The focus of these algorithms is on exploiting the structure and the sparsity pattern of the unconstrained MPC and MHE problems at the algorithm level, since the underlying linear algebra routines are dense.

About the choice of the algorithms, two structure-exploiting factorizations of the KKT matrix of the unconstrained MPC and MHE problems are presented. Namely, the backward Riccati recursion and the forward Schur-complement method are reviewed, and their efficient implementation in HPMPC is presented in details. None of these methods is novel in the field of MPC, but the level of performance obtained by the routines in HPMPC has not been obtained before, to the best of my knowledge. Furthermore, a collection of algorithms for condensing of MPC and MHE problems is presented. Some of the algorithms are well known, some are novel (to the best of my knowledge). Structure-exploiting factorizations and condensing methods find a conjunction point in partial condensing, that is a technique recently proposed to trade-off the horizon length and the input size in MPC and MHE problems [18].

Part III of the thesis briefly deals with algorithms for linear (constrained) MPC and MHE problems. In particular, two algorithms are presented: an IPM and an ADMM (Alternating Direction Method of Multipliers), both employing the backward Riccati recursion as a routine to solve tailored systems of linear equations. The focus is on showing that the special structure of the constraints can be exploited to efficiently handle them.

1.3 Thesis outline

Chapter 1 contains the introduction, comprising background, thesis approach, thesis structure and publication list. Afterwards, the thesis is divided into three parts.

Part I deals with efficient implementation methods for dense linear algebra routines, tailored for embedded optimization applications.

Chapter 2 states assumptions about the structure of the computer architectures considered in this thesis. Afterwards, it contains a brief review of the main approaches employed in the implementation of optimized BLAS libraries and in the implementation of linear algebra routines in embedded optimization. Finally, it compares the computational performance of the more promising options

currently available for the implementation of the dense Cholesky factorization for small to medium matrices.

Chapter 3 proposes a novel implementation method for level 3 BLAS and LAPACK routines (that are the backbone of algorithms for KKT matrix factorization and Hessian condensing), specially tailored for embedded optimization applications. The chapter begins with some assumption about the nature of the embedded optimization problems, and the consequences for the implementation of linear algebra routines. Afterwards, it presents a set of optimization techniques that provides good performance for small matrices, and it proposes a novel matrix format, that guarantees optimal performance of level 3 BLAS routines for matrices roughly fitting in last level cache. The step-by-step optimization of a key routine shows the performance impact of each single implementation technique. Finally, a selection of common level 3 BLAS and LAPACK routines is compared on two recent computer architectures, and the proposed approach is compared with the best proprietary and open-source BLAS libraries.

Chapter 4 is analogue to Chapter 3 but focusing on level 2 BLAS routines (that are the backbone of algorithms for KKT matrix solution, gradient condensing and residuals computation).

Chapter 5 introduces a number of common computer architectures, and describes in details the optimization of the general matrix-matrix multiplication `gemm` kernel on the different architectures, in both single and double precision.

Chapter 6 contains the summary of the Part I, and adds the Cholesky factorization routine implemented using the proposed methods to the initial performance tests performed in Chapter 2. Furthermore, it investigates if the use of code generation can further improve performance.

Part II contains a collection of structure-exploiting methods for the solution of unconstrained (linear) MPC and MHE problems. These methods are implemented using the efficient linear algebra routines proposed in Part I.

Chapter 7 introduces the unconstrained MPC and MHE problem formulations considered in the remainder of the part, and shows the structure of the KKT matrices for these problems.

Chapter 8 presents two structure-exploiting factorizations for the KKT matrix of the unconstrained MPC and MHE problems: a backward Riccati recursion and a forward Schur-complement recursion. Implementation of these routines using both optimized BLAS libraries and the custom linear algebra routines proposed in Part I are presented in details. Finally, exhaustive tests compare the performance of the two recursions, when implemented using optimized BLAS libraries or the proposed custom linear algebra routines.

Chapter 9 presents a wide collection of condensing methods for the MPC and MHE problems. All algorithms are initially presented for the MPC problem, and afterwards adapted to the MHE problem by considering the free initial state as an extra input variable at stage -1. In particular, three Hessian condensing methods (with one novel to the best of my knowledge), two Hessian factorization methods (with one novel to the best of my knowledge) and two Hessian solution methods (with one novel to the best of my knowledge) are presented. Furthermore, two combined methods for the simultaneous condensing and factorization of the Hessian matrix are presented by combining two Hessian condensing methods with the novel Hessian factorization method (one of the resulting algorithms is novel the best of my knowledge). The asymptotic complexity of all methods is analyzed both as number of flops and as execution time (with linear algebra routines implemented as described in Part I).

Chapter 10 reviews the idea of partial condensing and proposes efficient algorithms for the computation of the state space equations and cost function matrices and vector in the partially condensed MPC and MHE problems. The algorithms are based on the best algorithm for the Hessian condensing of the MHE problem, as investigated in Chapter 10. Furthermore, the backward Riccati recursion of Chapter 8 is used as an example to derive theoretical guidelines on the best value for the horizon length in the partially condensed MPC and MHE problems. Finally, the influence of the performance of linear algebra routines is investigated, finding that partial condensing gives additional performance advantages in case of very small matrices (to the best of my knowledge, this has not been investigated in literature yet).

Chapter 11 tailors the backward Riccati recursion and the novel Hessian condensing and solution methods to the special case of MPC problems with time-invariant matrices and time-variant vectors, in both the state space equations and the cost function expression, in the special case of the matrix of the terminal cost initialized to the solution of the discrete Riccati algebraic equations. Situations where problems in this form arise are described; in particular, problems in

this form arise as sub-problems in splitting methods for the constrained linear quadratic regulator. Problems in this form can be solved extremely efficiently, since the backward structure-exploiting factorization gives matrices constant over the stages. Therefore, there is no need to perform the factorization over all the stages, and only the matrices of a single stage need to be stored, greatly decreasing the memory requirements for the algorithms (these considerations are, to the best of my knowledge, novel).

Part III deals with algorithms for constrained (linear) MPC and MHE problems, that are implemented using the solvers from Part II as routines. Only algorithms for MPC are explicitly considered, the algorithms for MHE being analogue.

Chapter 12 introduces the linear (constrained) MPC formulations considered in the remainder of the part. In particular, hard and soft box constraints and general polytopic constraints are considered.

Chapter 13 presents two Riccati-based optimization methods for the solution of linear MPC problems: an IPM and an ADMM. These optimization methods have been widely used in literature, and have been chosen here in that they can benefit from an efficient implementation of the backward Riccati recursion. Furthermore, techniques to exploit the special structure of the soft constraints are presented, and numerical tests confirm that linear MPC problems with soft constraints can be solved in only slightly more time than the hard-constrained counterparts. The resulting IPM solvers are found to be more than an order of magnitude faster than a successful state-of-the-art solver for embedded optimization on a widely used benchmark, for medium to large problems. Furthermore, the overhead of the high-level wrapper is well amortized over the IPM iterations.

Chapter 14 contains a summary of Part III, and it shows the performance gains obtained using the efficient Riccati-based IPM solver and the forward Schur-complement recursion as routines in the real-time solution of challenging NMPC and NMHE problems using ACADO.

Appendix A proposes a modification to the open-source compiler `gcc` to improve the performance of the intrinsics for the various fused multiply-subtraction

instructions (employed e.g. in the implementation of the Cholesky factorization) in the x86_64 FMA ISA.

1.4 Publications list

The following journal articles ('article') and conference proceedings ('paper' or 'abstract') were published during the project period.

Paper [33] deals with algorithms for the solution of the unconstrained MPC problem, that make use of optimized BLAS and LAPACK routines in OpenBLAS [94] for the linear algebra. The first part of the paper reports some results originally found in the MSc thesis [31]: the backward Riccati recursion is found to be the solution method performing better for the widest range of problem sizes, in case of dense matrices in the state space equation and cost function formulation. The second part of the paper proposes techniques to improve the speed of a Riccati-based solver, such as the use of the Cholesky factorization to reduce the flop count (as shown in Chapter 8), or the use of mixed precision computation [23] (that is orthogonal to the techniques presented in this thesis).

Paper [77] presents an efficient IPM for the LPs arising in economic MPC of linear systems. The IPM combines a homogeneous and self-dual model with a specialized Riccati recursion, and it is tested on a power system management test problem. The subject of the paper is no further considered in this thesis.

Paper [35] compares techniques to parallelize the Cholesky factorization routine in the implementation of the backward Riccati recursion. In particular, the performance of the parallel version of OpenBLAS [94] is compared to the performance of PLASMA [13], that is a library providing LAPACK-like routines where multiple threads are explicitly handled in the linear algebra algorithms instead of in the level 3 BLAS routines. Furthermore, the asyncronous version of PLASMA allows for the threads of linear algebra routines to be scheduled asyncronously, while explicit barriers are employed to ensure correctness of the results. The subject of the paper is no further considered in this thesis, since the focus in on single-thread code.

Paper [34] proposes structure-exploiting condensing methods for the solution of the unconstrained MPC problem. In particular, two Hessian condensing

algorithms are considered, together with a novel structure-exploiting Hessian factorization and Hessian solution algorithms. If the novel Hessian solution algorithm is employed, there is no need to explicitly build the Hessian matrix, if the aim is the solution of the unconstrained MPC problem. The combination of one of the Hessian condensing algorithms with the structure-exploiting Hessian factorization algorithm and the explicit built of the Hessian matrix results in the Riccati-based algorithm originally proposed in [19]. The paper forms the basis of Chapter 9.

Paper [79] investigates the use of warm-starting to reduce the number of iterations of the Riccati-based homogeneous and self-dual IPM proposed in [77] for the solution of LPs in economic MPC of linear systems. The subject of the paper is no further considered in this thesis.

Paper [32] deals with a fast Riccati-based IPM for the solution of the linear MPC problem. The paper proposes to merge the backward Riccati factorization and the backward Riccati substitution in a single recursion, such that the matrices of the factorization are merged with the vectors of the substitution. This improves performance for small-scale problems. Furthermore, the paper proposes the use of custom linear algebra routines to improve the performance for small-scale problems, implemented using techniques such as blocking for cache and explicit vectorization. This represent the first iteration of the implementation strategy of Part I, while the Riccati-based IPM is considered in Chapter 13.

Paper [76] presents a Riccati-based ADMM for MPC with input and input-rate constraints, embedding an IPM for the efficient handling of the rate constraints at each ADMM iteration. The subject of the paper is no further considered in this thesis.

Paper [39] deals with a fast Riccati-based IPM for the solution of the linear MPC problem. In particular, it investigates the performance improvements that can be obtained exploiting the higher FP throughput of computations in single-precision, in conjunction with inexact search direction in the IPM and mixed precision computation. These techniques are not further considered in this thesis, mainly because the results in the paper are obtained with an early version of the linear algebra routines, that has not been updated yet in the single-precision case.

Paper [38] reviews computer architectures commonly employed in embedded optimization, and describes in details the optimization of the `gemm` kernel on these architectures. The the Riccati-based IPM implemented using these optimized linear algebra kernels is compared with a successful state-of-the-art solver for embedded optimization, and found to be several times faster. The review of computer architectures and the detailed description of the `gemm` kernel optimization forms the basis of Chapter 5, where many more computer architectures are added.

Abstract [36] deals with IPM and ADMM for MPC problems, tailored to efficiently handle soft constraints. In particular, the cost per iteration is only slightly larger than in the hard-constrained counterparts. The material for this abstract is now part of Chapter 13.

Paper [37] investigates the use of ARMv7A in MPC. The first part of the paper reviews the steps that lead to the implementation techniques proposed in Part I of this thesis. In particular, the paper introduces the panel-major matrix format. Afterwards, the computational capabilities of ARMv7A processors are investigated in detail, and the performance of the `gemm` and `gemv` kernels is evaluated. Finally, the performance of both a Riccati-based IPM and a Riccati-based ADMM are evaluated with respect to a successful state-of-the-art solver for embedded optimization. The material of the paper contributed mainly to Part I of the thesis.

Article [88] aims to assess the computational performance of NMPC and NMHE of a large-scale mechatronic application, namely the rotational startup of Airbone Wind Energy systems. The NPMC and NMHE problems are handled by ACADO, that employs either a condensed formulation of the QPs (solved by the AS qpOASES) or a sparse formulation of the QPs (solved by the IPM in HPMPC) arising as sub-problems in the NMPC problem. The material of the paper contributed to Chapter 14.

Paper [40] deals with the use of the forward Schur-complement recursion as a structure-exploiting solver for unconstrained MHE problems with additional equality constraints on the last stage. The implementation of the solver using the custom linear algebra routines in HPMPC is presented in details. Furthermore, the solver is tested in both a performance scalability test and as a routine in the solution of a challenging NMHE problem using ACADO (that is the same

application in [88]). The material of the paper contributed mainly to chapters 8 and 14.

Article [78] is the journal version of paper [77] and paper [79]. The subject of the article is no further considered in this thesis.

Article [57] investigates reformulations of step-response models as state-space models to solve them using the efficient block-factorization algorithms commonly employed in MPC. Furthermore, it proposes an IPM based on a backward Riccati recursion and a condensing method specially tailored to the shape of the step-response models formulated as state-space models. The subject of the article is no further considered in this thesis.

Part I

Dense Linear Algebra Routines for Embedded Optimization

Chapter 2

Review of dense linear algebra implementation techniques

This chapter presents a brief review of dense linear algebra implementation techniques. The reviews is not meant to be exhaustive, but simply to present some of the most widespread solutions for linear algebra implementation in the field of embedded optimization and control.

2.1 Assumptions about the computer architecture

In this thesis, only CPUs (Central Processing Units) are considered. Other processor types such as GPUs (Graphical Processing Units, and in particular GPGPUs, standing for General Purpose GPUs) and FPGAs (Field Programmable Gate Arrays) are not considered, even if their use in MPC is reported in literature [41, 24]. Nevertheless, some of the implementation techniques presented in later chapters may be applied to these processor types too.

The structure of modern CPU architectures can be better understood by putting them into historical perspective [84]. The performance of CPUs has improved exponentially for two decades, starting from early '80s. Until early 2000's, most of this performance burst came from increase of processor frequency. With the reduction in transistor size at each new lithography generation, the transistor could be operated at higher frequencies, and more transistor could be obtained from the same piece of silicon, driving the cost-per-transistor down. Code written and compiled for an older architecture could automatically run faster on more recent ones.

However, around 2004 CPUs hit the wall of power dissipation: the transistor got too dense to be able to dissipate the heat generated when operated at high frequencies, as the power consumption grows approximately with the square of the frequency. The most famous example is the one of the Intel NetBurst architecture: designed to reach 10 GHz, it could not increase frequencies beyond 4 GHz. Nonetheless, the empirical law about the doubling in transistor density every 18-24 months (known as Moore's law) has kept true up to today.

The solution to keep scaling performance without increasing frequencies is to have more work done per clock cycle. This has been achieved by exploiting the (still holding) reduction in transistor size and cost at each new lithography generation. Therefore, new CPUs architectures could use additional transistors to keep increasing performance by making cores wider (increase single-thread performance thankt to e.g. design of superscalar processors able to issue multiple instructions per cycle, or vector execution units) and by increasing the number of cores in the CPU. In both cases, code written and compiled for older architectures does not necessarily run faster on more recent ones. Furthermore, even recompilation can help to a limited extent, since the increased complexity of CPUs makes it difficult for the compiler to fully exploit hardware capabilities.

In this thesis, code is written to run on a single CPU core. General implementation techniques that can be used to write high-performance linear algebra routines on modern architectures are presented. In linear algebra routines design, the following architectural characteristics are assumed:

- the processor has a FP (Floating-Point) unit, or in general the cost (in clock cycles) of a memop (moving data from main memory to registers or vice versa) is higher than the cost of a flop (a floating-point operation).
- the FP unit is pipelined, meaning that the instruction throughput (the number of clock cycles between the beginning of the execution of an instruction and the beginning of the execution of the following (equal) instruction) is smaller than the latency (the number of clock cycles needed

to have the result of an instruction execution available for another operation).

- the FP unit may be able to operate simultaneously on small vectors of FP numbers thanks to SIMD (single-instruction multiple-data).
- there is at least one cache, with a LLC (last level cache) large enough to individually contain each data matrix within the size of interest.
- a cache line is large enough to contain several FP numbers.
- there is a MMU (memory management unit), with a TLB (translation lookaside buffer) used to cache virtual memory translations.

These assumptions hold true for virtually all reasonably recent desktop and mobile CPUs, and for many embedded CPUs.

Low-end embedded CPUs may lack cache or FP units: in this case, not all of the presented implementation techniques may be profitable. In particular, if there is not a FP unit, FP operations can be implemented in software (slow), or fixed-point operations have to be employed: this rises rather different issues and is not investigated in this thesis.

2.2 Linear algebra routines in high-performance computing: optimized libraries

This section contains a brief review of widespread libraries for linear algebra routines developed in the field ot high-performance computing.

2.2.1 Reference BLAS and LAPACK

BLAS (Basic Linear Algebra Subprograms) provides de-facto the standard building block for dense linear algebra routines. The original version was written in Fortran in 1979. The Netlib [5] version of the library is used as a reference, and is not optimized for speed.

BLAS is divided into 3 levels [42]:

- Level 1 BLAS contains routines for vector-vector operations, that perform $\mathcal{O}(n)$ flops on $\mathcal{O}(n)$ elements. Each vector element is reused $\mathcal{O}(1)$ times,

and typically exactly once (so there is no reuse of data). Since the time needed to move a vector element from main memory to register is generally much bigger that the time needed to perform a flops, level 1 BLAS routines are typically memory-bounded and with little room for optimization.

- Level 2 BLAS contains routines for matrix-vector operations, that perform $\mathcal{O}(n^2)$ flops on $\mathcal{O}(n^2)$ elements. Each matrix element is reused $\mathcal{O}(1)$ times, and typically exactly once (so there is no reuse of matrix data). On the other hand, each vector element is reused $\mathcal{O}(n)$ times. Therefore also level 2 BLAS routines are typically memory-bounded and with little room for optimization, that can partially reduce memory bandwidth requirements by re-using vector elements once in registers or cache. In Netlib BLAS, level 2 BLAS routines are implemented as simple nested double-loops.

- Level 3 BLAS contains routines for matrix-matrix operations, that perform $\mathcal{O}(n^3)$ flops on $\mathcal{O}(n^2)$ elements. Since each matrix element is reused $\mathcal{O}(n)$ times, well optimized versions of these routines can typically attain a large fraction of the full FP throughput by carefully reusing data once loaded into registers and caches. In Netlib BLAS, level 3 BLAS routines are implemented as simple nested triple-loops, without taking architectural parameters such as cache size into consideration.

In code making use of all BLAS levels, level 3 BLAS routine account for most of the computational cost. Therefore, most effort is spent in the optimization of these routines, also because in level 1 and 2 BLAS there is little space to improve routine performance. There exist several optimized implementation of BLAS, both commercial (e.g. Intel's MKL [12], AMD's ACML [3]) and open-source (e.g. ATLAS [4], GotoBLAS/OpenBLAS [10, 94], BLIS [6]).

LAPACK (Linear Algebra PACKage) is a library for numerical linear algebra [15]. It contains routines for more complex linear algebra operations such as factorizations, matrix inversions, solution of systems of linear equations, and least-square, eigenvalues and singular value problems. It is build on top of BLAS, as BLAS routines are used as much as possible in the implementation of LAPACK routines. LAPACK does not explicitly handle parallelization, and therefore it relies on an efficient BLAS implementation to attain high-performance on SMT (Simultaneous Multi-Thread) processors. LAPACK is written in Fortran 90 and can be found on the Netlib website [11], where the latest version is currently 3.5.0. Optimized BLAS implementations often provide optimized routines for key LAPACK routines such that factorizations and matrix inversions.

2.2.2 ATLAS

The ATLAS (Automatically Tuned Linear Algebra Software) project [91] is an instantiation of the AEOS (Automated Empirical Optimization of Software) paradigm. It provides an optimized implementation of BLAS that typically is much faster than the reference BLAS from Netlib.

The main idea is that new hardware architectures are released on a regular basis, and that is is too difficult or time consuming for a human operator to keep BLAS updated and optimized for each architecture. Therefore, ATLAS provides a framework to automatically generate an optimized BLAS library. The library depends on a number of parameters to adapt to the different architecture features, such as number of registers and cache size. During installation, the performance is automatically and empirically tuned on the specific machine by performing an optimization over the parameter space. The optimization often makes use of heuristics to reduce the size of the parameter space, possibly resulting in a sub-optimal choice of parameters.

The code of the original library is written entirely in C and depends on compilers to exploit different ISAs (Instruction Set Architectures). Recent versions often employ hand-optimized assembly kernels for performance critical routines in order to improve performance and explicitly target different ISAs. ATLAS employs implementation techniques such as block for registers and for different levels of cache, copy of data into aligned memory, and a block-wise matrix format (if matrices are large enough to justify the copy). The original `gemm` kernel is used to multiply squared sub-matrices fitting in L1 cache, where the left operand is transposed and the right is not-transposed. This scheme optimizes the memory access in case of scalar instructions, but it is not effective in case of SIMD instructions (present nowadays on most architectures).

ATLAS performance shows a large improvement with respect to reference BLAS, even if it is often not competitive with respect to hand-optimized BLAS libraries. On the other hand, ATLAS can quickly give reasonably good results on new hardware architectures.

2.2.3 GotoBLAS / OpenBLAS

A rather different approach is used in the GotoBLAS library [44]. In this implementation, the focus in on using analytical insight about architecture details to choose relevant architecture parameters.

One of the key differences in the implementation framework with respect to ATLAS is the focus on minimizing TLB (Translation Lookaside Buffer) misses [43], that is something ignored in previous BLAS implementation, that focused solely on caches. Furthermore, instead of blocking for L1 cache such that square sub-matrices of A, B and C can fit in L1 cache at once (as in ATLAS), GotoBLAS approach employs a multi-layered approach where the sub-matrices are not necessarily squared nor have the same size. A sub-matrix of the B matrix is kept into L1 cache while a bigger sub-matrix of A is streamed from L2 cache, one panel (i.e. sub-matrices where one dimension is big and the other is small) at a time. Registers are used to hold a small sub-matrix of C and therefore reusing elements from A and B once on registers, such that the memory bandwidth between L2 cache and registers is large enough to hold the stream of data. Furthermore, registers and software prefetch are employed to hide the higher latency of memory access from L2 cache compared to L1 cache. TLB misses are minimized by carefully rearranging data in memory such that elements are stored contiguously in the same order as they are accessed by the `gemm` kernel, and by considering also TLB size when choosing blocking size for L2 cache. The computationally most expensive part of the code (the 'inner-kernel') is hand-written in optimized assembly, explicitly targeting the ISA of the different architectures. The GotoBLAS inner-kernel consist of the three innermost loops of a layered approach, and it is therefore relatively complex.

GotoBLAS is typically faster than ATLAS and competitive with vendor's implementations: its performance is usually very close to the full FP throughput. The approach proposed in GotoBLAS is currently the best publicly-known approach to optimize BLAS for large-scale performance. GotoBLAS is no more under development, but a fork, OpenBLAS [94], provides optimized BLAS for recent architectures. In this thesis, the latest available version of OpenBLAS is employed (0.2.15).

2.2.4 BLIS

A recent effort to simplify the development of high-performance BLAS implementations is BLIS (BLAS-like Library Instantiation Software) [86]. It aims at providing a framework to quickly develop BLAS libraries for new architectures by focusing on code-reuse and portability [85].

BLIS simplifies GotoBLAS's approach by splitting the inner-kernel in two: the micro-kernel (i.e. the innermost loop) and a portable macro-kernel (consisting of two loops around the micro-kernel). The micro-kernel computes a sub-matrix of C by using two panels from A and B, and it is the only part of the code that needs to be carefully hand-optimized. Only one `gemm` variant is covered

by the micro-kernel, namely 'NT' (A not-transposed and B transposed): this is the optimal variant using SIMD instructions, since it avoids reductions and duplication operations in the innermost loop. This `gemm` micro-kernel is used to implement the entire level 3 BLAS by properly copying and transposing data matrices, and by using small routines for the corner cases. High-level algorithmic choices are taken from GotoBLAS as well, such as blocking for TLB and streaming from L2 cache.

All parameters in the BLIS implementation can be chosen by means of analytical insight [60]. be found in [9]. The BLIS approach makes the implementation much more clear than in the GotoBLAS approach, and therefore it also has an important pedagogical value. As a drawback, the current BLIS version focuses only on large-scale performance, and therefore the small-scale performance is particularly poor.

2.2.5 Intel's MKL

Intel Math Kernel Library (MKL) is a library of optimized mathematical routines, developed by Intel. It contains optimized versions of BLAS, LAPACK, ScaLAPACK, sparse solvers, fast Fourier transforms and vector mathematical routines. It is proprietary software, that under certain conditions can be redistributed as freeware. Each developer employing MKL requires a license; free academic license exists.

The BLAS version in MKL is generally considered to be the best option for Intel machines. In this thesis, the latest available version of MKL is employed (11.3).

2.3 Linear algebra routines in control: code generation

In optimized BLAS implementations, the focus is usually on large-scale performance, and small-scale performance can be poor due to the overhead of memory copy and unnecessary blocking. Therefore, BLAS is not often used in embedded optimization, since most problems in this field are of relatively small size.

Instead, an approach that has been widely employed in software for embedded optimization is code generation of linear-algebra routines. It exploits knowledge from the specific problem instance to generate a solver tailored to the special problem structure and size. Since the structure and size of each matrix is known,

this knowledge can be employed to perform optimizations at both generation and compilation time.

In the following, two widely employed approaches are reviewed. For the purposes of this thesis, code generation is solely considered as a linear algebra implementation techniques.

2.3.1 CVXGEN

Code generation for embedded optimization gained widespread attention thanks to CVXGEN [62]. CVXGEN is a code generator for convex optimization problems, and it has been widely employed in model predictive control. The optimization algorithm is a predictor-corrector IPM.

The search direction is computed by means of a sparse LDL factorization. Knowledge about problem structure (e.g. sparsity pattern) and size is used to fully unroll all loops in a Netlib-style triple-loop implementation of linear algebra routines and remove all unnecessary operations (e.g. multiplications by zero). All indexes are precomputed at generation time, and there are no branches in the code. CVXGEN then relies on the compiler to optimize the generated code for the target hardware. The main disadvantage of this approach is that the code size grows with the size of the problem size (cubically if the data matrices are dense, since all triple loops are fully unrolled), and therefore it is feasible only for very small and sparse problems. As a further drawback, this approach does not make use of instruction cache (since there is no code reuse), further penalizing performance.

In general, this approach avoids overhead of branches and reduces the number of flops by removing unnecessary operations, but it makes an extremely bad use of the hardware resources, attaining a very low performance when measured in Gflops.

2.3.2 FORCES

FORCES [26] is a code generator for numerical optimization, designed with special focus on optimal control problems. The optimization algorithm is a predictor-corrector interior point method. For the solution of the unconstrained sub-problems, it makes use of a block-wise Cholesky factorization of a block tridiagonal matrix, where generally all blocks have equal size $n_x \times n_x$.

It distinguishes between dense and sparse operations. Sparse operations are handled similarly to the CVXGEN solver. However, FORCES makes use of a rather different approach to code generation of dense linear algebra operations. Namely, dense linear algebra routines are implemented using Netlib-style triple-loops, but loop sizes are fixed and hard-coded at code-generation time. In this way, the compiler can decide to unroll loops where profitable. The main drawback of this approach is that it completely relies on the compiler for the code optimization: even if loop sizes are fixed, compilers are usually not able to properly optimize the code (as shown in Chapter 3), and thus this approach can typically attain only a small fraction of the full FP throughput. Another drawback is that, in case of different variable sizes at different stages of the optimal control problem, a different linear algebra routine has to be generated for each variable size, limiting code reuse to the case of identical operand sizes.

In general, this approach can outperform CVXGEN, but the performance in Gflops in only slightly better that the reference Netlib BLAS implementation, and far off the full FP throughput.

2.4 Comparison of existing dense linear algebra implementations

This section contains a brief comparison of linear algebra implementations of interest in embedded optimization. Not all existing implementations are tested, since results in literature can be used to select the most promising ones.

The test problem is the computation of the lower triangular Cholesky factor of a positive definite dense matrix. In double precision, this operation can be computed by means of the `dpotrf` routine in LAPACK. LAPACK makes use of level 3 BLAS for the most computationally intensive operations, and therefore this test indirectly evaluates also the performance of some BLAS routine. The `dpotrf` routine is important in embedded optimization since it is often used in structure-exploiting algorithms.

Reference BLAS and LAPACK version 3.5.0 are tested. The performance of the reference BLAS is found to be rather sensitive to the choice of compiler flags. The best performance with the `gcc` 4.8.4 compiler is obtained using the compiler flags `-O3 -funroll-loops`, plus the flags to enable all supported instruction sets on the test machines. Among the optimized open-source libraries, OpenBLAS is chosen for tests. ATLAS is disregarded since the GotoBLAS's approach (employed in OpenBLAS) is reported to give better performance [44]. BLIS is not considered since it is a reformulation of the GotoBLAS's approach

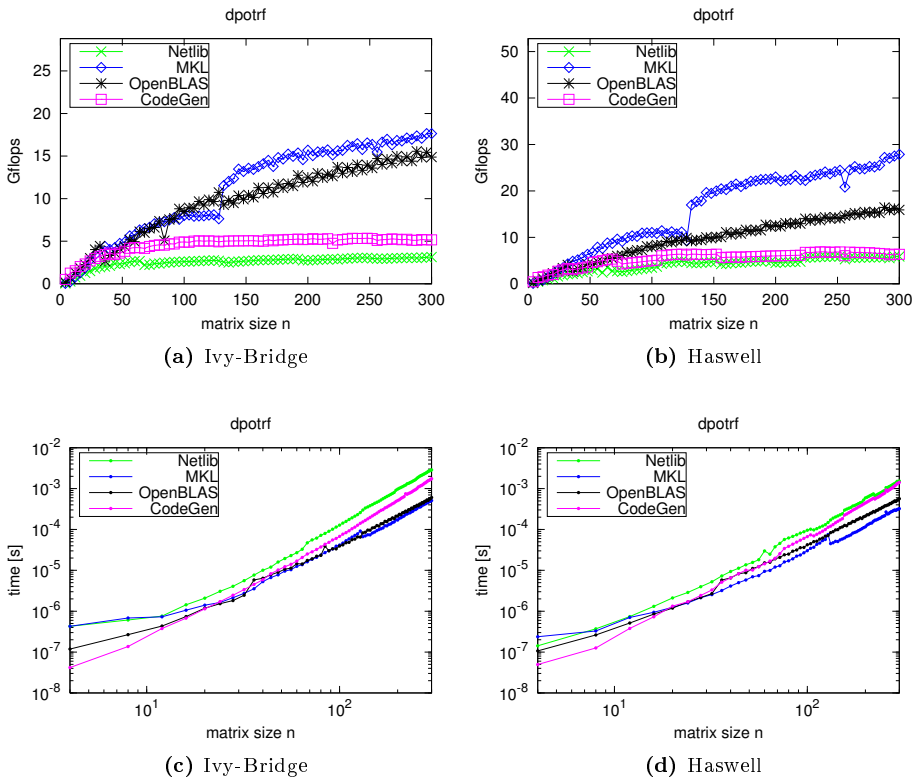
and it is still under active development and not completely optimized: therefore it provides lower performance, especially for small matrices. Additionally, MKL is considered, as it is the best proprietary library on Intel machines.

Regarding the code generation approaches, only FORCES's approach is considered. It consists on fixing at compile time the size of the triple-loop based linear algebra, such that the compiler can perform additional optimizations such as branch removal and loop unrolling. The code has been prepared by merging in a single routine and translating to C the needed parts (e.g. only lower triangular Cholesky factorization is considered) of the reference routines `dpotf2`, `dgemv`, `ddot`, `dscal` in BLAS and LAPACK. The code is compiled with `gcc` 4.8.4 with flags `-O3 -funroll-loops`, plus the flags to enable all supported instruction sets on the test machines (same configuration as the reference BLAS). CVXGEN's approach has not been considered, since it aims at sparse routines, and since it is clearly sub-optimal in relation to code size and instruction cache use in case of dense matrices.

The test machines are two laptops equipped with the processors Intel Core i7 3520M @ 3.6 GHz max turbo frequency (Ivy-Bridge micro-architecture) and Intel Core i7 4800MQ @ 3.7 GHz max turbo frequency (Haswell micro-architecture). Only single-thread code is considered in this test (and more generally in this thesis). The Ivy-Bridge micro-architecture supports the AVX ISA, that is enabled in `gcc` with the `-mavx` flag. The Haswell micro-architecture supports the AVX2 and FMA ISAs (that are enabled in `gcc` with the `-mavx2 -mfma` flags), and has double the full FP throughput when these new ISAs are exploited. These machines will be extensively employed for test in this thesis, and more details about them will be given in later sections.

The results of the tests are in Figure 2.1. The bottom plots report the computation time in seconds, in logarithmic scale, averaged over a large number of operations to increase the accuracy. The top plots report the performance of the routines in Gflops (that is, billions of FP operations per second), that is computed by dividing the flop count (computed considering the leading term in the computational complexity, i.e. $\frac{1}{3}n^3$ flops for the `dpotrf` routine) by the time in seconds. Performance plots are scaled such that the full FP throughput (28.8 Gflops on the Ivy-Bridge processor and 52.8 on the Haswell processor) is at the top of each picture. As a note, the full PF throughput for the Haswell processor assumes a frequency of 3.3 GHz, that is the maximum frequency the processor can operate at, in case of code employing 256-bit wide instructions.

The reference BLAS version (x) steadily gives a rather low performance, many times lower than the full FP throughput. The code-generated version (□) of the reference BLAS improves performance in case of very small matrices, but as the matrix size increases, the performance gap with respect to the reference BLAS



(a) Ivy-Bridge (b) Haswell

(c) Ivy-Bridge (d) Haswell

Figure 2.1: Performance test for the LAPACK `dpotrf` routine on an Intel Core i7 3520M processor (Ivy Bridge micro-architecture, supporting the AVX ISA) and an Intel Core i7 4800MQ (Haswell micro-architecture, supporting the AVX2 and FMA ISAs).

version vanishes. This hints at the fact that the compiler can produce similar code in both cases, and that the performance advantage of the code-generated version comes from a lower number of branches and smaller overhead in function calls, rather than from structural differences in the code. Despite the fact that the Haswell processor has double the full FP throughput, the performance is only slightly higher than on the Ivy-Bridge processor.

Both optimized BLAS implementation perform slightly worse than the code-generated reference version for small matrices (with a cross-over point for matrix size around 20), but considerable better for larger ones, and the performance gap increases with the matrix size. It is interesting to notice that the OpenBLAS ($*$) version does not give better performance on the Haswell processor, hinting at the fact that the code has not been optimized yet for this architecture. On the contrary, the performance of the MKL version ($\diamond$) nearly doubles on the Haswell processor.

This test is a good example of the fact that nowadays FP throughput improves mainly due to more and wider execution units, since the CPU frequency have almost stalled. However, it requires a considerable amount of work to exploit the computational capabilities of the new hardware, since generally compilers (as the `gcc` version 4.8.4 in these tests) are not able to compile generic code (as e.g. the reference BLAS code) into high-performing routines, not even if code-generated. Additionally, code optimized for an older architecture does not necessary perform better on a new one (as e.g. `dpotrf` in OpenBLAS 0.2.15).

2.5 Conclusion

In this section, some dense linear algebra implementations are reviewed. The most promising ones are compared using the Cholesky factorization as a test case.

The main results of the tests are:

- Compilers are often unable to compile generic triple-loop based linear algebra code into high-performance routines. This was shown for the `gcc` compiler, but it is true more generally. The performance of the resulting routines is very sensitive to the choice of compiler flags.
- Optimized BLAS libraries employ advanced implementation techniques that target specific architectural parameters. The code optimized for an

architecture does not necessarily run faster on a more recent one (e.g. `dpotrf` in OpenBLAS 0.2.15).

- Code-generation of triple-loop based linear algebra can improve the performance only for very small matrices, but the resulting routines are structurally analogue to the original reference ones. The performance advantages come from fewer branches and lower functions overhead. As such, code-generation could be associated with other implementation techniques to improve performance for very small matrices, at the cost of requiring the generation of code for each problem instance. Therefore, code-generation is an implementation technique orthogonal to the ones presented in the remainder of the thesis.

Chapter 3

Level 3 BLAS and LAPACK for embedded optimization

Level 3 BLAS contains routines for basic matrix-matrix operations. LAPACK contains routines for more complex linear algebra operations such as factorizations and matrix inversions, and it makes use of level 3 BLAS routines. In the embedded optimization framework, level 3 BLAS and LAPACK routines can be used in the implementation of second order optimization methods.

If n is the size (meaning the number of rows or columns) of all matrices involved in a generic level 3 BLAS or LAPACK operation, then the computational cost of the routines is of about $\mathcal{O}(n^3)$ flops, that is cubic in the matrix size. On the other hand, the data size in memory is of $\mathcal{O}(n^2)$ elements, that is quadratic in the matrix size. This means that each matrix element is used $\mathcal{O}(n)$ times. In modern computers, the time needed to move a matrix element from main memory into registers is much larger than the time needed to perform a floating-point operation on that element once in registers. Therefore, the reuse of matrix elements in registers and caches is a key requirement to obtain high-performance level 3 BLAS and LAPACK routines.

In the implementation of all level 3 BLAS routines, most of the computation can be cast in terms of the `gemm` routine, that is the general matrix-matrix multiplication routine. In the optimization of level 3 BLAS routines for large

scale matrices, a carefully optimized `gemm` routine can be used to obtain high-performance implementations of all other level 3 BLAS routines [45]. Therefore, the `gemm` routine is often used as a benchmark for BLAS implementations.

This chapter is organized as follows. Section 3.1 presents the characteristic features of embedded optimization problems, that are exploited in the remainder of the chapter to design level 3 BLAS routines specially tailored to this class of problems. Section 3.2 presents in details the optimization of the key routine in level 3 BLAS: the `gemm`. Section 3.3 presents the optimization of other level 3 BLAS and LAPACK routines. Section 3.4 presents the comparison of different implementation techniques for selected level 3 BLAS routines. Finally, Section 3.5 contains the performance plot for some widely used level 3 BLAS and LAPACK routines.

3.1 General framework: embedded optimization

Problems in embedded optimization often must be solved in real-time on resource-constrained hardware. This poses challenges on the development of fast-enough solvers.

Linear algebra routines are a key aspect in the implementation of these solvers, since they perform the most computationally expensive operations. A set of linear algebra routines specially tailored for embedded optimization problems can take advantage of the special features of this class of problems in order to reduce the computational time. The following features are considered:

1. Embedded optimization problems must often be solved in real-time on resource-constrained hardware. The computational speed is a key factor.
2. The size of the matrices is generally relatively small, i.e. with size n in the order of tens or a few hundreds.
3. Each data matrix is often reused several times, e.g. to solve similar optimal control problems at each sampling instant, or to solve similar unconstrained optimization problems at each iteration of an IPM.
4. Structure-exploiting optimization algorithms can exploit the high-level sparsity pattern of the problem, and therefore the data matrices are generally dense.

These features can be exploited in the design of the linear algebra routines as follows:

1. Linear algebra routines must make an efficient use of available hardware resources. Compilers are generally unable to convert generic triple-loop based linear algebra source code into efficient routines fully exploiting hardware capabilities. This is due to the fact that modern computer architectures are increasingly complex, and compilers lack additional information about the aim of the routines and the kind of data they are designed to handle. Therefore the programmer should consider high-level information into the routines design, as well as explicitly target advanced hardware features as e.g. pipelines and vector execution units.

   This excludes the use of triple-loop based linear algebra routines, that are commonly employed in MPC.

2. Matrices with size n in the order of tens or a few hundreds are assumed to fit in some cache level. As a consequence, implementation techniques like blocking for different cache levels are not considered, simplifying the design of the linear algebra routines. Furthermore, for small matrices the cost of copying or scaling matrices is not negligible with respect to the cost of performing level 3 BLAS operations. Therefore, linear algebra routines should be designed to reduce as much as possible the need to copying or scaling matrices.

   This excludes the use of existing BLAS implementations, since they are optimized for large scale matrices.

3. Since data matrices are often reused several times, it makes sense to store them in a matrix format that is particularly favorable for the linear algebra routines. The cost to convert matrices into this format can be amortized over several matrix reuses, or the conversion may even be performed off-line in some cases.

   This excludes as well the use of existing BLAS implementations, since they assume matrices to be stored in column-major (or Fortran-like) or row-major (or C-like) orders, while internally using optimized formats.

4. Sparse linear algebra requires the use of a special matrix storage and the efficient handling of the matrix element indexes. The use of sparse linear algebra makes sense only if the matrix elements are scattered over the matrix, and not gathered into dense sub-matrices, otherwise dense linear algebra on these sub-matrices is preferable. Sparse linear algebra can not make use of processor features as e.g. vectorization, and therefore it has a low performance with respect to the full FP throughput, and it makes sense only in case of really sparse problems. Therefore, only dense linear algebra routines are considered, with the exception of very special and common sparse matrices with fixed structure (i.e. diagonal, triangular).

3.2 Optimizing the gemm routine

The gemm routine is the general matrix-matrix multiplication routine. In the BLAS standard, it has the interface (considering the double precision version, and using C notation)

```
void dgemm_(char *transA, char *transB, int *m, int *n, int *k, \
    double *alpha, double *A, int *lda, double *B, int *ldb, \
    double *beta, double *C, int *ldc);
```

and it computes

$$C \leftarrow \alpha \cdot \mathrm{op}(A) \cdot \mathrm{op}(B) + \beta \cdot C$$

where α and β are scalars, and $\mathrm{op}(A)$ can be either A of A^T (similarly for B) depending on the flags `transA` and `transB`. The matrices $\mathrm{op}(A)$, $\mathrm{op}(B)$ and C have size $m \times k$, $k \times n$ and $m \times n$. The matrices A, B and C are stored in column-major (or Fortran-like) order, where consecutive elements on the same row are stored `lda` (standing for leading dimension of matrix A), `ldb` and `ldc` positions away in memory: therefore these quantities represent the number of rows in the matrices as allocated in memory. This gives great flexibility in the computation with sub-matrices.

The following alternative interfaces are considered in this thesis:

```
void dgemm_nn_lib(int m, int n, int k, double *A, int sda, \
    double *B, int sdb, int alg, double *C, int sdc, \
    double *D, int sdd, int tc, int td);
```

computing

$$\mathrm{op}(D) \leftarrow \alpha \cdot A \cdot B + \beta \cdot \mathrm{op(C)}$$

and

```
void dgemm_nt_lib(int m, int n, int k, double *A, int sda, \
    double *B, int sdb, int alg, double *C, int sdc, \
    double *D, int sdd, int tc, int td);
```

$$\mathrm{op}(D) \leftarrow \alpha \cdot A \cdot B^T + \beta \cdot \mathrm{op(C)}$$

where op(C) and op(D) are controlled by the flags `tc` and `td`, and α and β are controlled by the `alg` flag as

$$\texttt{alg} = \begin{cases} 0 & \Rightarrow \quad \alpha \leftarrow 1, \beta \leftarrow 0 \\ 1 & \Rightarrow \quad \alpha \leftarrow 1, \beta \leftarrow 1 \\ -1 & \Rightarrow \quad \alpha \leftarrow -1, \beta \leftarrow 1 \end{cases}$$

These are the only cases (corresponding to computation, upgrade and downgrade of the result matrix) that needed to be implemented, and they are explicitly implemented avoiding therefore to scale the result sub-matrices. In the 'NN' ('NT') version, the matrices A, B, op(C) and op(D) have size $m \times k$, $k \times n$ ($n \times k$), $m \times n$ and $m \times n$. The matrices A, B, C and D are assumed to be stored in the panel-major format proposed in Section 3.2.2. The quantities `sda`, `sdb`, `sdc` and `sdd` represent the 'secondary' dimension of matrices in panel-major format, i.e. the number of columns in the matrices as allocated in memory.

Notice that these routines take 4 matrix operands, meaning that the result matrix D does not necessarily overwrite the matrix C: it can do so if `C` and `D` correspond to the same memory location. This feature is useful in many cases to avoid an explicit matrix copy.

Compared to the standard `gemm` interface, these alternative formulations are somehow lower-level interfaces, closer to the interface of the underlying `gemm` kernels. This has the advantage of reducing the overhead of the routine, increasing performance for small matrices. As a drawback, they are less general, covering only the cases commonly encountered in embedded optimization.

3.2.1 Optimizing the gemm kernel

The `gemm` kernels are the routines accounting for the innermost loop in the implementation of the different `gemm` variants. These kernels compute a fixed-size sub-matrix of the result matrix D by adding a fixed-size sub-matrix of C with the product of two panels (meaning with this a matrix where one of the two dimension is much larger than the other) from A and B, where one of the two panel dimensions is fixed. Each iteration of the kernel loop computes a rank-1 update of the result sub-matrix. In order to achieve the best performance, the `gemm` kernels are optimized for different computer architectures. Therefore features as e.g. the height of the panel and the size of the fixed-size sub-matrix of D computed by the kernels are architecture-dependent. As an example, the 'NT' variant of the `gemm` kernel computing a 8×4 sub-matrix of D, where the matrices are assumed to be stored in the panel-major format with panel height $b_s = 4$ (see Section 3.2.2 for more details), is

```
void kernel_dgemm_nt_8x4_lib4(int kmax, double *A, int sda, \
    double *B, int alg, double *C, int sdc, double *D, int sdd, \
    int alg, int tc, int td);
```

Notice that the interface of the gemm routines closely resemble the interface of the corresponding underlying gemm kernel.

The gemm kernels are the key kernels in all level 3 BLAS routines, since the computationally most expensive parts of all level 3 BLAS routines can be cast in term of these kernels. In turn, LAPACK routines are build on top of level 3 BLAS routines, and therefore the gemm kernels account for most of the computations in LAPACK routines too. This section presents the generic techniques used in the implementation of all gemm kernels, and shows how to optimize them for an hypothetical computer architecture. A collection of gemm kernels optimized for a number of computer architectures and the specific implementation details are in Chapter 5.

3.2.1.1 Blocking for registers

The most important technique is probably blocking for registers, meaning with that the simultaneous computation of all elements of a sub-matrix of the result matrix small enough to fit into registers. It has the twofold aim of hiding latency of instructions, and reducing the number of memops.

About hiding instructions latency, most FP instructions are pipelined, and their latency is larger than their throughput. A pipelined instruction is performed on a number of steps, each one taking a certain amount of clock cycles (often 1) and being performed by a different circuitry. The operands of the instruction have to go through all the steps, one after the other, and the result is available after a number of clock cycles equal to the latency of the instruction. The idea of pipelining is that, while an instruction is at a certain stage of the pipeline, other equal but independent instructions (meaning a sequence of the same instruction operating on independent operands, such that the result of an instruction is not the operand of another) can be processed at the same time, at other stages of the pipeline. If all instructions are independent, after an initial delay needed for the first instruction to go through the entire pipeline (and equal to the latency of the instruction), every number of clock cycles equal to the throughput another instruction is completed, and at any time all stages of the pipeline are busy, working on different operands. If there is dependency between the output of an instruction and the input of another instruction, then the second instruction can not be processed until the result of the first instruction is available: this stalls the pipeline.

Blocking for registers can be used to hide latency of instructions to obtain full throughput, since the computation of several result elements at the same time can provide enough independent instruction to keep the pipeline full.

Blocking for registers can also be used to reduce the number of memops, since each matrix element can be reused several times once loaded into registers: this means that fewer loads are necessary to perform the same number of flops. This is useful to reduce the memory bandwidth requirements below the maximum memory bandwidth available in the system, and therefore to avoid that the kernel becomes memory-bounded.

The idea is explained with an example. Let us consider an hypothetical processor that can perform a FMA (fused-multiply-add) every clock cycle (throughput=1), while the result is available after 4 clock cycles (latency=4). The FMA is a 3-operands instruction defined as

$$z \leftarrow \mathrm{FMA}(x, y, z) \doteq z + x \cdot y.$$

Furthermore, that hypothetical processor can load one FP register from L1 cache every clock cycle.

Without loss of generality, the following gemm operation is considered: two squared matrices A and B of size $n \times n$ are multiplied, and the result is used to update the content of a square matrix C of size $n \times n$, as

$$C \leftarrow C + A \cdot B.$$

Using the definition of matrix-matrix product, each element c_{ij} of C can be computed as the inner product

$$c_{ij} = c_{ij} + \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}.$$

that is performed using the sequence of n FMA instructions

$$\begin{array}{c|c} & b_{kj} \\ \hline a_{ik} & c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj} \end{array}, \quad k = 0, 1, \ldots, n-1. \tag{3.1}$$

as

$$\begin{aligned} c_{ij} &\leftarrow c_{ij} + a_{i0} \cdot b_{0j} \\ c_{ij} &\leftarrow c_{ij} + a_{i1} \cdot b_{1j} \\ c_{ij} &\leftarrow c_{ij} + a_{i2} \cdot b_{2j} \\ &\ldots\ldots \end{aligned}$$

where the colors are used to highlight instruction dependencies. These FMA instructions are dependent, because c_{ij} is the result of an instruction and the operand of the following one. Therefore, a FMA can be performed every 4 clock cycles. In any case, 2 FP numbers (a_{ik} and b_{kj}) need to be loaded from memory to perform each FMA. Therefore also ignoring for a moment the latency constraint, it would not be possible to perform 1 FMA every clock cycle and keep the pipeline full due to the memory bandwidth constraint.

One way to keep the pipeline full besides these constraints is to compute several elements of the matrix C at the same time. If 4 FP registers can be used to hold a 2×2 sub-matrix of C, then the 4 elements of the sub-matrix can be computed simultaneously (using 0 and 1 as indexes in place of $i+0$, $i+1$, $j+0$, $j+1$ to keep the notation lighter)

$$\begin{array}{c|cc} & b_{k0} & b_{k1} \\ \hline a_{0k} & c_{00} \leftarrow c_{00} + a_{0k} \cdot b_{k0} & c_{01} \leftarrow c_{01} + a_{0k} \cdot b_{k1} \\ a_{1k} & c_{10} \leftarrow c_{10} + a_{1k} \cdot b_{k0} & c_{11} \leftarrow c_{11} + a_{1k} \cdot b_{k1} \end{array} \quad , \quad k = 0, \ldots, n-1. \tag{3.2}$$

as

$$\begin{aligned} c_{00} &\leftarrow c_{00} + a_{00} \cdot b_{00} \\ c_{10} &\leftarrow c_{10} + a_{10} \cdot b_{00} \\ c_{01} &\leftarrow c_{01} + a_{00} \cdot b_{01} \\ c_{11} &\leftarrow c_{11} + a_{10} \cdot b_{01} \\ c_{00} &\leftarrow c_{00} + a_{01} \cdot b_{10} \\ c_{10} &\leftarrow c_{10} + a_{11} \cdot b_{10} \\ c_{01} &\leftarrow c_{01} + a_{01} \cdot b_{11} \\ c_{11} &\leftarrow c_{11} + a_{11} \cdot b_{11} \\ c_{00} &\leftarrow c_{00} + a_{02} \cdot b_{20} \\ &\ldots\ldots \end{aligned}$$

Regarding the instruction latency issue, this time in the 3 idle clock cycles between consecutive updates of the c_{00} element, other 3 elements are updated, ideally keeping the pipeline full if the elements from A and B can be loaded fast enough.

Regarding the memory bandwidth issue, each A and B element is reused 2 times once in registers, since e.g. in the update of c_{10} only a_{1k} needs to be loaded, since b_{k0} has already been loaded to compute c_{00} at the previous clock cycle, and so on. Therefore, only 1 element from either A or B needs to be loaded at each clock cycle. Generally speaking, in level 3 BLAS operations, a cubic

number of flops is performed on a quadratic number of matrix elements, so the larger the sub-matrix held in registers, the higher the reuse-factor.

Thanks to the benefits of hiding instruction latency and decreasing memory bandwidth requirements, it is possible to keep the FMA pipeline full and get full throughput while satisfying the latency and bandwidth constraints. The achieved speed-up with respect to the case (3.1) is of a factor 4.

The blocking idea can be applied to other memory levels (as for example blocking for level 2 or 3 cache) to take into account the fact that the available memory bandwidth decreases at lower levels in the memory hierarchy. However, since our target are relatively small scale matrices that are assumed to fit in some cache level, blocking for cache is not further considered in this thesis.

3.2.1.2 Use of SIMD instructions

SIMD (Single-Instruction Multiple-Data) are instructions that perform the same operation in parallel on all elements of small vectors of data. In theory, instructions operating on vectors of size n_v can improve the performance up to a factor n_v.

Many modern architectures have SIMD instructions, since they are a relatively easy and efficient way to increase single-thread performance, especially in scientific computing. In particular, x86 and x86_64 architectures have several versions of SSE instructions (operating on 128-bit-wide vectors, each holding 2 double or 4 single precision FP numbers) and AVX instructions (operating on 256-bit-wide vectors, each holding 4 double or 8 single precision FP numbers), while ARM architecture has NEON instructions (operating on 128-bit-wide vectors).

Compilers can attempt to automatically vectorize scalar code, emitting SIMD instructions. However, this is not a simple task, since the use of SIMD may require deep changes to the code structure (e.g. to ensure proper alignment of data) that are better suited to the programmer understanding of the overall algorithm. The use of SIMD can be ensured by explicitly coding them in assembly or inline assembly (low level solution, that gives full control also over the instruction scheduling and register allocation) or by means of intrinsics (higher lever solution, where intrinsics are special functions called from C code and directly mapped to SIMD instructions, leaving to the compiler instruction scheduling and register allocation).

Continuing the example, let us assume that the hypothetical processor has 2-

wide vector units (i.e. each holding 2 FP numbers), and vector FMA and load instructions (operating on the 2-wide vectors) with the same latency and throughput of the scalar instructions used in Section 3.2.1.1. The kernel operating on the 2×2 sub-matrix in (3.2) can be implemented using the vector instructions as

	b_{k0}	b_{k1}
a_{0k} a_{1k}	$\begin{bmatrix} c_{00} \\ c_{10} \end{bmatrix} \leftarrow \begin{bmatrix} c_{00} \\ c_{10} \end{bmatrix} + \begin{bmatrix} a_{0k} \\ a_{1k} \end{bmatrix} \cdot \begin{bmatrix} b_{k0} \\ b_{k0} \end{bmatrix}$	$\begin{bmatrix} c_{01} \\ c_{11} \end{bmatrix} \leftarrow \begin{bmatrix} c_{01} \\ c_{11} \end{bmatrix} + \begin{bmatrix} a_{0k} \\ a_{1k} \end{bmatrix} \cdot \begin{bmatrix} b_{k1} \\ b_{k1} \end{bmatrix}$

(with $k = 1, \ldots n-1$), where the squared brackets indicates the small vectors. As a result, the number of instructions is halved. This means that there are no more enough independent instructions to keep the vector FMA pipeline full, and the throughput is equal to the one in (3.2), since a 2-wide FMA every 2 clock cycles is equivalent to 1 scalar FMA per clock cycle. It is important to note that the vectors containing elements from B have the same element repeated: depending on the ISA, this has to be done explicitly (e.g. using shuffle or broadcast instructions in SSE and AVX ISAs), or it is handled implicitly by the FMA instruction (e.g. NEON ISA).

In order to have again enough independent instructions to keep the pipeline full, it is possible to compute an additional 2×2 sub-matrix of C. Therefore, the optimal `gemm` kernel making use of vector instructions computes e.g. a 4×2 (or possibly a 2×4, even if this has the disadvantage of requiring more shuffles and broadcast instructions, due to the fact that the elements from B are repeated) sub-matrix of C as

	b_{k0}	b_{k1}
a_{0k} a_{1k}	$\begin{bmatrix} c_{00} \\ c_{10} \end{bmatrix} \leftarrow \begin{bmatrix} c_{00} \\ c_{10} \end{bmatrix} + \begin{bmatrix} a_{0k} \\ a_{1k} \end{bmatrix} \cdot \begin{bmatrix} b_{k0} \\ b_{k0} \end{bmatrix}$	$\begin{bmatrix} c_{01} \\ c_{11} \end{bmatrix} \leftarrow \begin{bmatrix} c_{01} \\ c_{11} \end{bmatrix} + \begin{bmatrix} a_{0k} \\ a_{1k} \end{bmatrix} \cdot \begin{bmatrix} b_{k1} \\ b_{k1} \end{bmatrix}$
a_{2k} a_{3k}	$\begin{bmatrix} c_{20} \\ c_{30} \end{bmatrix} \leftarrow \begin{bmatrix} c_{20} \\ c_{30} \end{bmatrix} + \begin{bmatrix} a_{2k} \\ a_{3k} \end{bmatrix} \cdot \begin{bmatrix} b_{k0} \\ b_{k0} \end{bmatrix}$	$\begin{bmatrix} c_{21} \\ c_{31} \end{bmatrix} \leftarrow \begin{bmatrix} c_{21} \\ c_{31} \end{bmatrix} + \begin{bmatrix} a_{2k} \\ a_{3k} \end{bmatrix} \cdot \begin{bmatrix} b_{k1} \\ b_{k1} \end{bmatrix}$

(3.3)

(with $k = 1, \ldots n-1$). As a side effect, each B element is now used 4 times, further reducing memory bandwidth requirements. Overall, this kernel gives a speed-up of a factor 2 with respect to (3.2), and therefore of a factor 8 with respect to (3.1).

A drawback of vectorization is that a kernel operating on larger sub-matrices of C is used. This means that, as the vector size increases, it also increases the minimum size of the sub-matrix required to get full throughput, and therefore the performance increases get smaller for small matrices. Generally speaking, large latencies and low memory bandwidth have a negative effect on performance for small-scale matrices.

As said, SIMD often have alignment requirements. In our implementation, this

is automatically ensured by the choice of the matrix format used by the linear algebra routines.

3.2.2 Use of contiguous memory and panel-major matrix format

The use of contiguous memory is important for several reasons: it helps to fully exploit the available memory bandwidth, it improves cache reuse and it reduces the TLB misses.

When an element is fetched from memory, data is moved into cache in chunks (called cache lines) of typically 32 or 64 bytes. This means that the access to elements belonging to the same cache line is faster, since only one cache line needs to be moved into cache. On the contrary, random access of elements often requires a different cache line for each element. Therefore the access of contiguous elements maximizes the effective memory bandwidth.

In order to speed-up cache access and reduce its complexity and cost, a certain cache line can be mapped in a limited number n of locations in cache: this kind of cache is called n-way associative. As a consequence, it may happen that cache lines are evicted from cache even if this is not fully utilized. As an example, if a matrix is stored in column-major (or Fortran-like) order, for certain column length it can happen that contiguous elements on the same row are mapped into the same cache location, evicting each other. This effectively acts as a reduction in cache size. Use of contiguous memory can mitigate this, since consecutive cache lines are mapped in different cache locations.

Finally, memory is seen from a program as virtual memory, that is mapped into physical memory locations by means of a translation table in the MMU (Memory Management Unit), the page table. The TLB (Translation Lookaside Buffer) is a cache for the page table, containing the physical address of the most recently used memory pages (each usually of size 4 KB). If memory is accessed in a non-contiguous way, it may happen that TLB is not large enough to translate the entire content of cache, increasing the number of expensive TLB misses.

In [43], a `gemm` design based on the minimization of the TLB misses is proposed. In this approach, the needed sub-matrices from the A and B matrices are packed into memory buffers before each call to the `gemm` kernel. These sub-matrices are carefully packed (and possibly transposed) using a multi-layered matrix format. Matrix elements are stored in the exact same order as accessed by the `gemm` kernel, and taking into account cache and TLB sizes. This approach gives near full FP throughput for large matrices, but it incurs in a notable overhead for

small matrices, since in this case the (quadratic) cost of packing data dominates the (cubic) cost of performing FP operations.

Taking into account the fact that matrices in embedded optimization are relatively small, and therefore assumed to fit in cache, it is possible to modify the above approach to reduce the overhead due to the packing of data. The key idea is that data matrices in embedded optimization are often reused several times. Therefore it makes sense to convert only once the data matrices into an optimal format (used as the default matrix format by all linear algebra routines) and reuse the converted matrices several times, therefore well amortizing the conversion cost. Furthermore, linear algebra routines can be designed such that the output matrix is automatically stored into this optimal format at no extra cost, meaning that only the original data matrices possibly need to be converted.

Since blocking for cache is not employed, the optimal matrix format is rather simple: the complex matrix format proposed in [43] simplifies to a single layer. Namely, in the `gemm` routine, the A and B matrices are packed into horizontal panels of contiguous data, as showed in Figure 3.1. The panel height (in the following b_s, for block size) has to be the same for all operand matrices. As a consequence, the generic kernel size $m_r \times n_r$ has the constraint that both m_r and n_r have to be multiple of b_s. The values of m_r and n_r are architecture-dependent and a function of the number of registers as well as the SIMD width. The value of b_s is usually chosen as the smaller of m_r and n_r, such that every time a cache line is accessed, it is fully utilized.

Fig. 3.1 shows the panel-major matrix layout and the behavior of the 'NT' variant of the `gemm` micro-kernel, that computes $D \leftarrow \alpha \cdot A \cdot B^T + \beta \cdot C$, where the left factor A is not-transposed and the right factor B is transposed. This is the optimal variant, since both A and B are accessed panel-wise (i.e. data is read along panels). Furthermore, the regular access pattern of data in memory (i.e. access of contiguous memory locations) can be easily detected by the hardware prefetcher (if present in the architecture). On the contrary, in the 'NN' variant the A matrix is optimally accessed panel-wise, but the B matrix is accessed across panels (i.e. only a few columns of each B panel are used, before moving to the following panel), therefore making a worse use of caches and TLBs. This complex access pattern is generally not detected by the hardware prefetcher, and therefore software prefetch has to be explicitly used to move B elements into cache before they are needed. In summary, embedded optimization algorithms making use of the proposed linear algebra routines should be designed to use the 'NT' `gemm` variant whenever possible.

Continuing the example of the hypothetical processor, since the SIMD width is 2 and the optimal `gemm` kernel is 4×2, a good choice for b_s is 2. In Fig. 3.1 the behavior of the 4×2 kernel operating on matrices packed with $b_s = 2$ is shown:

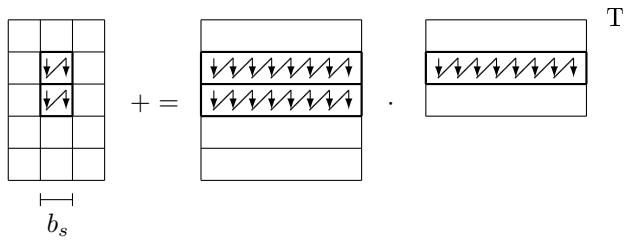


Figure 3.1: Matrix layout in memory (called panel-major matrix format): matrix elements are stored in the same order such as the gemm micro-kernel accesses them. This micro-kernel implements the optimal 'NT' variant (namely, A not-transposed, B transposed). The panel height b_s is the same for the left and the right matrix operand, as well as for the result matrix. Each arrow represents the b_s elements that are on the same column within a panel. The diagonal lines indicate that, once the last element of a column is accessed, the following element to be accessed is the first of the consecutive column within the same panel.

two panels of A and one panel of B are streamed to compute 4×2 elements of C. Notice that, in this case, following the approach proposed in [43], A would be packed into a buffer with $b_s = 4$ while B would be packed into a buffer with $b_s = 2$: this is the optimal choice since it further improves cache use and reduces TLB misses, but it is not compatible with the constraint that b_s has to be the same for all data matrices. This constraint is necessary to allow the use the panel-major matrix format as the common matrix format for all linear-algebra routines.

The proposed approach gives steady and near to full FP throughput for matrices fitting in LLC, minimizing the issues related to cache associativity and TLB misses. The performance is high also for small matrices, since the cost of packing data into the optimal matrix format is well amortized.

3.2.3 Order of outer loops

A gemm routine optimized for small matrices can be implemented by means of two loops around the carefully optimized gemm kernel. In case of a gemm kernel where m_r and n_r are not equal, the order of these two loops has a big impact on the performance of the gemm routine as the size of the factor matrices increase.

Let us assume that $m_r > n_r$ (i.e. the `gemm` kernel computes a sub-matrix of C with more rows that columns): this is the usual case in architectures with SIMD instructions, since it reduces the cost of shuffling the vectors containing the B elements (see Chapter 5 for more details). Therefore, in the `gemm` kernel, the number of streamed panels from A (i.e. m_r/b_s) is larger than the number of streamed panels from B (i.e. n_r/b_s). Under this assumption, it is convenient that the outermost loop is over the rows, and the intermediate loop over the columns of the result matrix. In this way, the B matrix is swept by loading n_r/b_s panels at a time from L2 cache or LLC while the m_r/b_s panels from A are kept into L1 cache. This reduces the memory bandwidth requirements from L2 cache or LLC, since a lower number of non-L1-residents panels needs to be loaded to compute the same amount of flops.

Ignoring cache associativity, as a rule of thumb this approach gives close to full performance in the computation of matrices with k up to the value such that $m_r \cdot k + n_r \cdot k$ elements can fit in L1 cache at once. In practice, this k value is often in the range 200 to 400, that is large enough for most embedded optimization applications. For larger values of k, optimal performance can be recovered by adding blocking for different cache levels. However, this is not of interest for this thesis.

3.2.4 Transposition, edges and corners handling

The interface of the `gemm` kernel has two flags to control the transposition of the C and D matrices. This is something rather different compared to the standard `gemm` interface in BLAS, that allows for the matrices A and B to be transposed. This different choice is justify by the fact that the proposed linear algebra routines are specially tailored for small scale performance, and therefore a lower level interface (i.e. closer to the interface of the underlying `gemm` kernel) has been chosen in order to reduce overhead. In the `gemm` kernel, the transposition of the C and D matrices can be performed at little or no extra cost (depending on the ISA), and it adds extra flexibility. For example, the product $D \leftarrow A^T \cdot B^T$ can be rewritten as $D \leftarrow (B \cdot A)^T$ and therefore cast in terms of the 'NN' variant of the `gemm` routine.

About edges and corners handling, let us assume that the optimal `gemm` kernel has size $m_r \times n_r$. Since in general $m_r > 1$ and $n_r > 1$, there exists the issue of computing the result matrix C when m is not multiple of m_r and n is not multiple of n_r.

In optimized BLAS libraries where packing is employed, this is handled in the packing routine [45, 86]: the sub-matrices are padded with zeros while packed

in the buffers, in order to ensure the correctness of the result. Then the result sub-matrix of the exact size is copied from a buffer into the correct memory location, again ensuring correctness of the result. However, this approach can not be employed since in the proposed gemm implementation matrices are not packed into buffers, and in any case it introduces overhead due to the additional matrix copy.

A simple solution that does not require packing is to already add the padding to each matrix C when stored in memory in the panel-major format, such that the padded matrix C^+ has size $m^+ \times n^+$, where m^+ is the smaller multiple of m_r that is larger than m, and similarly for n^+. In this way it is possible to simply repeat the kernel until all the result matrix is fully covered. However, this approach has the severe drawback of preventing the work with sub-matrices, since the data surrounding the result sub-matrix may be corrupted.

Another solution would be the use of smaller kernels to exactly cover the edges and the corners of the result matrix. This requires a trade-off between performance and code reuse. In fact, it is possible to handle all cases with just 3 additional kernels (in the following called 'unitary' kernels), of size $m_r \times 1$, $1 \times n_r$ and 1×1, and repeat them until the edges and the corners are exactly covered. However, these kernels generally have very low performance: it is possible to partially alleviate the instruction latency constraint by using several registers to store partial accumulations of each result element, but it is not possible to overcome the memory bandwidth constraint. This affects particularly the performance of small matrices, that is the main focus in embedded optimization.

On the opposite side, it is possible to write a kernel to handle each edge and corner with just one kernel call. These kernels give better performance than the repetition of unitary kernels. However, the number of required kernels grows with the square of the sizes of the optimal gemm kernel size, making this approach unappealing due to the need of writing and testing a large number of kernels (lack of code reuse).

In the proposed linear algebra implementation framework, the most efficient solution is found to be the following. A few small kernels are designed: the number of rows is a multiple of the effective SIMD width (in fact, the time required to compute 1 element or an entire SIMD vector of elements is the same), or a power of two in case of scaler ISAs; the number of columns is a power of two. This greatly reduces the number of possible row and column size combinations. Each kernel is designed to compute in registers a sub-matrix of the result matrix equal to the kernel size, but to store in memory sub-matrices of different (equal or smaller) sizes. The overall set of kernels is able to store sub-matrices of any size equal or smaller than the optimal gemm kernel. An optimal kernel without the variable-storing-size logic is used for the sub-matrices entirely

in the inside of the result matrix C: this avoids the overhead of the storing logic. The variable-storing-size kernels take care of the corners, and at a computational cost lower that the repetition of unitary kernels.

As an example, the optimal 'NT' variant of the `gemm` kernel for the AVX instruction set computes a 8×4 sub-matrix of D, where the matrices are assumed to be stored in the panel-major format with panel height $b_s = 4$. The interface is:

```
void kernel_dgemm_nt_8x4_lib4(int kmax, double *A, int sda, \
    double *B, int alg, double *C, int sdc, double *D, int sdd, \
    int alg, int tc, int td);
```

The analogue kernel allowing for variable-storing-size has the interface

```
void kernel_dgemm_nt_8x4_vs_lib4(int mr, int nr, int kmax, \
    double *A, int sda, double *B, int alg, double *C, int sdc, \
    double *D, int sdd, int alg, int tc, int td);
```

where m_r is in the range 5 to 8, and n_r is in the range 3 to 4. Other kernels take care of the remaining cases.

3.2.5 Low rank updates handling

Low rank updates are matrix-matrix products where m and n are much larger than k, and therefore the products computes a large matrix with small rank.

The implementation techniques described in this section are highly effective in case the rank k is not too small. In fact, the `gemm` kernel is a loop over k, and therefore if k is very small, the overhead of calling the kernel, zeroing accumulation registers before the loop, shuffling accumulation registers after the loop and storing results can be easily much higher than the time spent on the loop itself. Therefore, the performance of the `gemm` routine can be poor in case of low rank updates, that are rather frequent operations (e.g. in the structure exploiting factorization of the condensed Hessian matrix presented in Chapter 9).

A possible solution to this is the implementation of specialized kernels for low rank updates, where the order of the two innermost loops is switched (i.e. the

innermost loop gets over n and the middle loop gets over k). Furthermore, the kernel performs a fixed-rank update of the result matrix, e.g. the cases of rank equal to 1, 2, 3 and 4 can be explicitly coded, and higher rank updates can be computed looping over the rank 4 kernel plus a final clean-up.

The advantage of this implementation is that the cases of rank 1, 2, 3 and 4 are explicitly coded, and therefore the `gemm` routine reduces to only two loops, while the loop over k is totally unrolled. This greatly improves the performance, at the expense of requiring a specialized kernel for each explicitly coded rank size.

The low rank implementation can therefore be employed if k is smaller than a certain threshold, while the standard implementation if k is larger. The same technique can be employed in the implementation of e.g. the `syrk` routine.

3.3 Optimizing other level 3 BLAS and LAPACK routines

The computationally most expensive parts of level 3 BLAS and LAPACK routines can be cast in terms of the `gemm` kernel [45]. In fact, the `gemm` kernel can be used to compute, upgrade or downgrade a rectangular sub-matrix of the result matrix with the product of two rectangular matrices.

The reminder of the section presents techniques to obtain high-performance level 3 BLAS and LAPACK routines based on the optimized `gemm` kernel, with special focus on small-scale performance.

3.3.1 Triangles, factorizations, substitutions and inversions handling

In the implementation of level 3 BLAS and LAPACK routines, the `gemm` kernel can not take care of triangular factor matrices, triangular result matrices, factorizations, substitutions (i.e. solution of triangular system of equations) and inversions. These specialized operations require specialized routines. Several approaches can be used in the implementation of these routines and in their use of the `gemm` kernel.

In standard BLAS and LAPACK, routine to handle triangular matrices and substitutions are part of level 3 BLAS, while factorizations and inversions are part of LAPACK, that is build on top of BLAS.

In optimized level 3 BLAS libraries, when packing is employed it is possible to implement all level 3 BLAS routines (with the exception of `trsm`, implementing substitutions) using the sole `gemm` kernel and proper packing/padding routines [45]. The `trsm` routine is an exception, since the downgrade part of the routine can be cast in terms of `gemm` kernel, while the substitution part can not. In [86], two `trsm` approaches are compared. In one approach, the `gemm` kernel is explicitly used for the downgrade, while another specialized routine (not a kernel, since there are no loops) takes care of the substitution part. This approach has the advantage of requiring the design only of the `gemm` kernel, but it has the drawback of larger overhead since there are two function calls and the result sub-matrix needs to be loaded and stored in memory twice. In the other approach, the `gemm` kernel and the specialized substitution routines are merged into a single `trsm` kernel. This requires the design of a specialized `trsm` kernel, but it has lower overhead and therefore it gives better performance for small matrices.

In the proposed implementation of linear algebra for embedded optimization, the second approach for the implementation of all level 3 BLAS routines is employed, since it gives the best performance for small matrices. Therefore, a specialized kernel (or better, a set of specialized kernels to cover all sizes of the result matrix, as done for the `gemm` kernel in Section 3.2.4) is designed. In such kernels, the main loop is literally copied-and-pasted from the `gemm` kernel, while specialized procedures before and after this loop take care of triangular matrices and substitutions. This approach requires the design of several specialized kernels, but once the `gemm` kernel is available, it can be easily edited to get all other level 3 BLAS kernels.

LAPACK routines make use for BLAS routines, but in general not of BLAS kernels, since their interfaces are not standardized and therefore not exposed (the BLIS project is an exception, exposing also its lower level interface). LAPACK contains both unblocked and blocked versions of all routines. Unblocked versions make use of level 2 BLAS and elementary operations such that square roots and divisions. They compute the result matrix one row or column at a time, and are usually employed for small matrices and as routines in blocked versions. Blocked versions make use of level 3 BLAS and unblocked LAPACK routines for factorizations and substitutions (that are the matrix equivalent of square roots and divisions). They compute the result matrix one sub-matrix at a time, and they rely on the underlying optimized BLAS routines to provide high-performance for large matrices. In the context of embedded optimization, the main drawback of this approach is that it suffers from a considerably overhead (due to the many levels of routines), and the small-scale performance is particularly poor.

Some optimized BLAS libraries (as e.g. OpenBLAS) contain an optimized ver-

sion of some of the key LAPACK routines (such as Cholesky and LU factorization, triangular matrix inversion, multiplication of two triangular matrices). These routine are written making use of the optimized level 3 BLAS kernels (and not routines), and therefore exhibit a much better performance for small matrices. In particular, this allows the choice of a much smaller threshold to switch to the blocked version of the algorithms, therefore casting more computations in the terms of the optimized BLAS kernels.

In the proposed implementation of linear algebra routines for embedded optimization, there is no distinction between level 3 BLAS and LAPACK routines. Namely, special kernels are written for the LAPACK routines as well, and they are implemented using the same approach used for all level 3 BLAS routines. Therefore, there are no unblocked LAPACK routines, and the optimized kernels are used for all matrix sizes. Said in another way, the block size of the blocked version of LAPACK routines is chosen equal to the `gemm` kernel size, and the unblocked version of LAPACK routines is merged with level 3 BLAS kernels to build specialized kernels. In case of small matrices, numerical tests show that this approach gives the best performance.

3.3.2 Merging of linear algebra routines

In the case of level 3 BLAS and LAPACK routines, the best performance for small matrices is given by the use of tailored kernels where both the main loop of the `gemm` kernel (accounting for the upgrade and downgrade) and the specialized procedures (handling triangular sub-matrices, and factorizations, substitutions and inversion of fixed-size matrices) are merged into a single kernel. The approach can be generalized to more complex operations, that are not part of BLAS or LAPACK but that can be computed using two or more BLAS and LAPACK routines. In some cases, specialized kernels can be written for these complex operations, that reduce the number of function calls and avoid the overhead of repeatedly loading and storing the same sub-matrix. In other cases, it may be possible to merger routines that internally make use of the same kernel (e.g. `potrf` and `trsm`), such that the merged routine operates on larger matrices, reducing the overhead due to the handling of edges and corners.

Some examples of complex operations that are commonly found in embedded optimizations and that can be easily merged are:

- (Symmetric) matrix upgrade followed by Cholesky factorization. This is a very common operation, that computes $D \leftarrow (C + A \cdot A^T)^{1/2}$. The upgrade of the C matrix (i.e. the computation of $A \cdot A^T$) and the downgrade in

the Cholesky factorization are both computed uinsg the main loop in the `gemm` kernel, and therefore they can be naturally merged. Two kernels are required, one for the computation of the diagonal blocks (`syrk_potrf` kernel, giving symmetric upgrade of C, symmetric downgrade and Cholesky factorization) and one for the off-diagonal blocks (`gemm_trsm` kernel, giving upgrade of C, downgrade and substitution). This routine can be used to efficiently implement the backward Riccati recursion in Chapter 8.

A variant of this operation is the case where the matrix A is triangular. The required kernels are `lauum_potrf` and `trmm_trsm` for the diagonal and off-diagonal blocks respectively. This routine can be employed in the implementation of the forward Schur-complement recursion in Chapter 8.

- Cholesky factorization followed by matrix substitution. This is another very common operation, that is e.g. used in the implementation of the Schur complement as $A \cdot B^{-1} \cdot A^T = A \cdot (L \cdot L^T)^{-1} A^T = (AL^{-T}) \cdot (AL^{-T})^T$, where B is a positive definite matrix and L is its lower triangular Cholesky factor. The factorization of B and the computation of AL^{-T} can be merged in a single routine, that can be implemented as a Cholesky factorization routine that operates on rectangular matrices (i.e. in the case of the lower triangular Cholesky factorization, for matrices with more rows that columns). This routine can be employed in the implementation of the forward Schur-complement recursion in Chapter 8.

- The explicit expression for the inverse of a lower triangular matrix can be obtained by considering the inversion of a 2×2 block partition of the matrix, as

$$\begin{bmatrix} A & \\ B & C \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} & \\ -C^{-1}BA^{-1} & C^{-1} \end{bmatrix}.$$

  This algorithm can be implemented in several fashions, either recursively or iteratively. The triangular matrix inversion is implemented in the LAPACK routine `trtri`, that makes use of an iterative algorithm. At a generic iteration, the sub-matrix A^{-1} contains the part of the matrix that has already been inverted, while C^{-1} has not been computed yet. Therefore, the new block $-C^{-1}BA^{-1}$ is computed using `trmm` for the operation $B \cdot A^{-1}$ (since A^{-1} is available explicitly), and `trsm` for the operation $C^{-1}(BA^{-1})$ (since C is available but C^{-1} is not).

  The proposed implementation computes the transposed inverse instead of the inverse: therefore the transposed inverse of a lower triangular matrix is an upper triangular matrix. This choice is justified by the fact that most of the computation in this operation can be cast in terms of the optimal 'NT' version of the `gemm` kernel. The explicit expression for the transposed inverse of a lower triangular matrix can be obtained by considering the

inversion of a 2×2 block partition of the matrix, as

$$\begin{bmatrix} A & \\ B & C \end{bmatrix}^{-T} = \begin{bmatrix} A^{-T} & -A^{-T}B^{T}C^{-T} \\ & C^{-T} \end{bmatrix}.$$

The routine is implemented in a fashion similar to the other BLAS and LAPACK routines, namely the result matrix is computed in sub-matrices of the size of the optimal `gemm` kernel, one panel at a time. In the computation of each new panel, a specialized routines computes the fixed-size sub-matrix of A^{-T}. In case of vector instructions available in the architecture, an effective way to compute A^{-T} is as IA^{-T}, i.e. by using a procedure for `trsm` on the identity matrix. Afterwards, the element on the top-left corner is computed using a specialized kernel `trtri`, that can be seen as a merge of a kernel for `trmm` (for the computation of $-A^{-T}B^{T}$, that internally makes use of the 'NT' version of the `gemm` kernel, since the matrix A^{-T} is directly available, and the matrix B is transposed by the kernel) and a kernel for `trsm` (that computes $(A^{-T}B^{T})C^{-T}$, where the sub-matrix C has fixed size).

In practice, these specialized routines are found to boost performance for small matrices, while somehow lowering performance for larger ones. This is due to the fact that these complex operations have more operands, and therefore the overall size of data matrices will exceed caches size for smaller size of the operand matrices.

3.3.3 Notable routines

This section reports the interface of notable routines as implemented in HPMPC, together with the rationale behind the choice of the interfaces. All matrices are assumed to be in panel-major format.

3.3.3.1 syrk

The `syrk` routine has the interface

```
void dsyrk_nt_lib(int m, int n, int k, double *pA, int sda, \
    double *pB, int sdb, int alg, double *pC, int sdc, \
    double *pD, int sdd);
```

and it computes the lower triangular part of the matrix $D \leftarrow \alpha \cdot A \cdot B^T + \beta \cdot C$. The interface is similar to the one of the `gemm` routine, with the exception of the lack of flags for transposition (that has not been implemented yet). The possibility of having two distinguished matrices for the right and the left factor can speed up the computation of the lower triangular part of the matrix $(X \cdot Y) \cdot X^T$, in case it is not favorable to compute the Cholesky factorization of Y to ensure symmetry.

3.3.3.2 potrf

The `potrf` routine has the interface

```
void dpotrf_lib(int m, int n, double *pC, int sdc, \
    double *pD, int sdd, double *inv_diag_D);
```

where the top $n \times n$ sub-matrix C_0 of the matrix C is supposed to be symmetric positive semi-definite. If $m = n$, the routine computes the lower triangular Cholesky factor D of the $n \times n$ matrix C. If $m > n$, the routine computes the lower triangular Cholesky factor D_0 of the upper $n \times n$ matrix C_0, and use it to compute the substitution D_1 of the lower $(m-n) \times n$ matrix C_1, as

$$\begin{bmatrix} D_0 \\ D_1 \end{bmatrix} = \texttt{dpotrf\_lib}\left(\begin{bmatrix} C_0 \\ C_1 \end{bmatrix}\right) = \begin{bmatrix} C_0^{1/2} \\ C_1 D_0^{-T} \end{bmatrix}$$

where the exponent $^{1/2}$ denotes the lower triangular Cholesky factorization. The $n \times 1$ vector `inv_diag_D` returns the inverse of the diagonal of the lower triangular Cholesky factor D, and it can be used to speed up the computation of subsequent substitutions, avoiding the need to compute divisions.

If the matrix is singular, when one of the diagonal elements is found to be zero during factorization, that element and all elements below on the same column are set to zero, as well as the corresponding element in the inverse diagonal vector. Therefore singularity can be detected by checking for zero elements in `inv_diag_D`. In this way, the routine can be used for the factorization of a (squared) symmetric positive semi-definite matrix, that is an operation useful to speed up the implementation of the backward Riccati recursion in case of singular recursion matrix.

3.3.3.3 syrk_potrf

The `syrk_potrf` routine is a merger of the `syrk` and `potrf` routines. Namely the routine has the interface

```
void dsyrk_dpotrf_lib(int m, int n, int k, double *pA, int sda, \
    double *pB, int sdb, int alg, double *pC, int sdc, \
    double *pD, int sdd, double *inv_diag_D);
```

It computes

$$\begin{bmatrix} D_0 \\ D_1 \end{bmatrix} = \texttt{dpotrf\_lib}\left(\alpha \cdot \begin{bmatrix} A_0 \\ A_1 \end{bmatrix} \cdot B^T + \beta \cdot \begin{bmatrix} C_0 \\ C_1 \end{bmatrix}\right) = \begin{bmatrix} (\alpha \cdot A_0 \cdot B^T + \beta \cdot C_0)^{1/2} \\ (\alpha \cdot A_1 \cdot B^T + \beta \cdot C_1) D_0^{-T} \end{bmatrix}$$

where the matrix A_0 has size $n \times k$, the matrix A_1 has size $(m-n) \times k$, the matrix B has size $k \times n$, the matrices C_0 and D_0 have size $n \times n$ and the matrices C_1 and D_1 have size $(m-n) \times n$.

3.3.3.4 lauum_potrf

The `lauum_potrf` routine is a merger of the LAPACK `lauum` and `potrf` routines. Namely the routine has the interface

```
void dlauum_dpotrf_lib(int m, int n, int k, double *pA, int sda, \
    double *pB, int sdb, int alg, double *pC, int sdc, \
    double *pD, int sdd, double *inv_diag_D);
```

It computes

$$\begin{bmatrix} D_0 \\ D_1 \end{bmatrix} = \texttt{dpotrf\_lib}\left(\alpha \cdot A \cdot B^T + \beta \cdot \begin{bmatrix} C_0 \\ C_1 \end{bmatrix}\right) = \begin{bmatrix} (\alpha \cdot A \cdot B^T + \beta \cdot C_0)^{1/2} \\ (\beta \cdot C_1) D_0^{-T} \end{bmatrix}$$

where the matrix A has size $k \times k$, the matrix B has size $k \times k$, the matrices C_0 and D_0 have size $n \times n$ and the matrices C_1 and D_1 have size $(m-n) \times n$.

3.3.3.5 trtri

The `trtri` routine is has the interface

```
void dtrtri_lib(int m, double *pA, int sda, int use_inv_diag_A, \
    double *inv_diag_A, double *pC, int sdc);
```

It computes the transposed inverse $C = A^{-T}$ of the matrix A. The choice to compute the transposed inverse instead of the inverse is justified by the fact that the former internally makes use of the 'NT' version of the `gemm` kernel (that is the optimal version), while the latter makes use of the 'NN' version. Furthermore, if the `trtri` routine is used in the inversion of a positive definite matrix, the choice of computing the transposed inverse implies that the result matrix is already in the correct format to be used as an input for a `lauum` routine implemented using the 'NT' version of `gemm`.

3.4 Comparison of implementation techniques for `dsyrk + dpotrf`

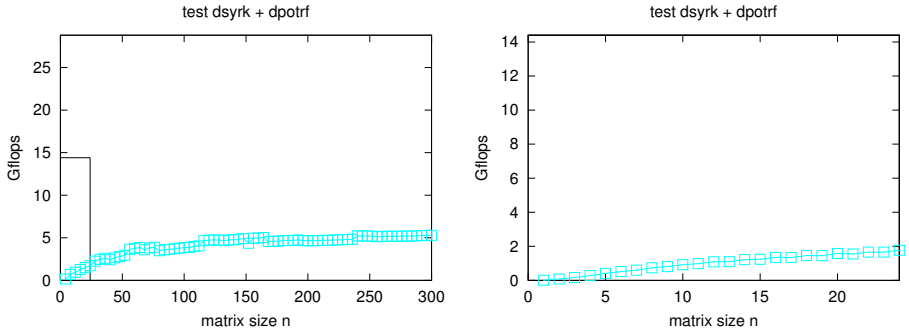
In this section contains an exhaustive comparison of implementation techniques for the `dsyrk + dpotrf` routine. Namely, the operation of interest is

$$\left(A + B \cdot B^T\right)^{1/2}$$

where A and B are squared matrices of size n, and A is symmetric positive definite. The exponent $^{1/2}$ indicates the lower triangular Cholesky factorization.

The test processor is an Intel Core i7 3520M, that during all tests runs at the maximum turbo frequency of 3.6 GHz. The processor implements the Ivy-Bridge micro-architecture and supports the SSE4.2 (that operates on 128-bit vectors, each containing 2 doubles) and AVX (that operates on 256-bit vectors, each containing 4 doubles) instruction sets. In particular, the AVX ISA is employed, since it gives twice the full FP throughput compared to the SSE4.2 ISA. The Ivy-Bridge core can perform a vector multiply and a vector add every clock cycle: this gives a full FP throughput of $2 \cdot 4 \cdot 3.6 = 28.8$ Gflops in double precision.

The operative system is an Ubuntu 14.04 distribution, and the Linux kernel version is 3.19. The code is compiled with `gcc` 4.8.4, and the use of the AVX instruction set is enabled by means of the `-mavx` flag. The flags `-O3` and `-funroll-loops` are used for the reference and code-generated reference implementations, where the latter flag is found to slightly improve performance. The flag `-O2` is used for OpenBLAS and for the proposed linear algebra for embedded optimization (HPMPC), since these implementations are already hand optimized, and perform loop unroll explicitly.

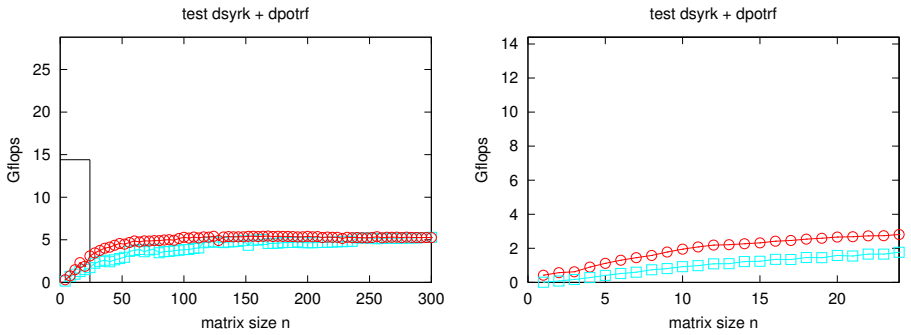


(a) (b)

Figure 3.2

The comparison is made as comments to a series of performance plots. For each figure, there are two sub-figures: one for matrix size n up to 300 in steps of 4 (left sub-figure), and one for matrix size up to 24 in steps of 1 (right sub-figure). The left sub-figure is scaled such that the top of the pictures corresponds to the full FP throughput (28.8 Gflops). The right sub-figure can be seen as a zoom into the black square in the left sub-figure, and it is scaled such that the top of the picture corresponds to 50% of the full FP throughput (14.4 Gflops).

Figure 3.2 shows the performance plot of the `dsyrk` routine from the reference Netlib BLAS and the `dpotrf` routine from LAPACK using the Netlib BLAS (□). Both routines are coded in Fortran. This approach gives a maximum performance of about 5 Gflops, that is about 18% of full FP throughput. The performance graph is steadily low for all matrix size: therefore there are likely no memory bandwidth limitations, but the processor sits idle most of the time due to instruction dependencies, or it performs unnecessary operations. The `gcc` compiler can be used to produce the assembly version of the compiled code. In this case, the inspection of the assembly code shows that the compiler is able to auto-vectorize the code (that therefore makes use of the AVX ISA), but that the structure of the code is not deeply changed (e.g. there is no sign of blocking for registers or cache). As a comparison, if the reference code is compiled without the `-mavx` flag or with the `-O2` flag, scalar instructions are emitted, and the maximum attained performance is slightly lower than 3 Gflops (about 10%). Therefore, even if vector instructions give a full FP throughput 4 times as high as scalar instructions, the performance given by employing vectorization alone is lower than twice as high, and still many times lower than full FP throughput.



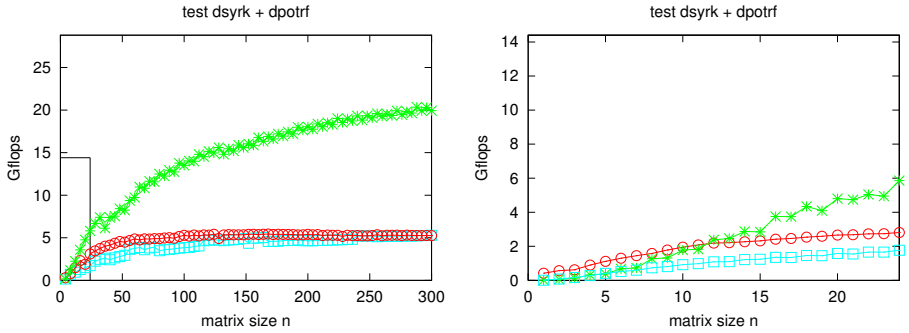
(a) (b)

Figure 3.3

Figure 3.3 adds the performance plot of the C translation of the reference routines presented in Figure 3.2 (o). Furthermore, the code generation approach used e.g. in the implementation of FORCES is employed to improve performance: the size of all loops is fixed at compile time. Therefore the compiler can perform additional optimizations such as remove branches and unroll loops when profitable. The performance graph is steadily for all matrix size, and shows a noticeable improvement only for very small matrices. As the matrix size increases, the performance plots of the reference and code-generated references implementations gets almost indistinguishable. Also this time the assembly code inspection shows that the compiler is able to auto-vectorize the code, but not to deeply change the structure of the code. As a consequence, the performance improvements for small matrices come from the reduction of overhead due to branches and function calls, rather than from structural improvements in the code.

Figure 3.4 adds the performance plot of the `dsyrk` and `potrf` as provided by OpenBLAS version 0.2.15 (*). OpenBLAS is an highly optimized BLAS implementation, that provides also optimized version of key LAPACK routines such as `dpotrf`. For large matrices, the performance of this implementation approaches the full FP throughput, arriving at 20 Gflops (or 70%) for the largest tested matrix size. Due to blocking for cache, the performance remains close to full FP throughput for even larger matrix sizes. For small matrix size, the performance is worse that both reference BLAS (break-even point around 5) and code-generated reference BLAS (break-even point around 12).

The asymptotic performance of OpenBLAS is so much better than the previous



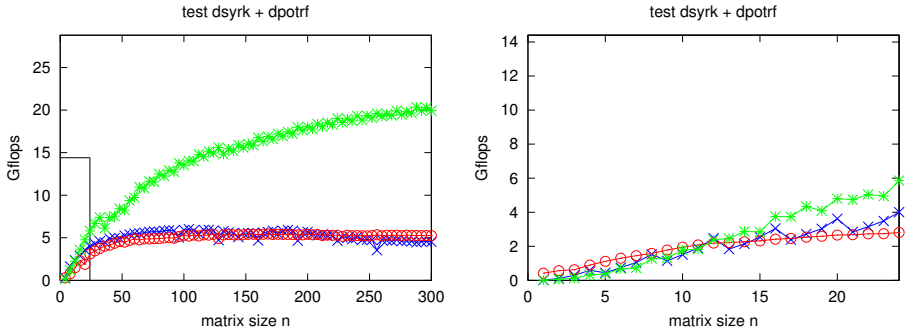
(a) (b)

Figure 3.4

implementations due to the fact that its code is tailored for the specific hardware. In particular, the tested version is optimized for the Sandy-Bridge architecture.

Figure 3.5 removes the performance plot of the reference `dsyrk` and `potrf` routines, and adds the performance plot of the first iteration of the proposed linear algebra for embedded optimization (x). This first iteration uses only C code (no assembly nor intrinsics), but it exploits knowledge about the number of available registers to perform block for registers. The matrices are stored in column-major (or Fortran) order. The performance is similar to the reference and the code-generated reference versions, arriving at a maximum of 5.8 Gflops (20% of full FP throughput). The assembly code inspections shows that in this case the compiler is not able to auto-vectorize the code, and therefore the attained performance is actually 80% of the scalar full throughput. This shows as the use of blocking for register alone gives as much speed up as the use of 4-wide vector instructions without blocking for registers. For large matrices, or for matrix size such that 96, 128, 160, 192, 256, the performance decreases due to memory bandwidth constraints or finite associativity of cache. The left sub-figures shows a stair-wise performance, with steps equal to 4: this is due to the fact that the optimal `gemm` kernel size is 4×2.

Figure 3.6 shows the performance plot of the second iteration of the proposed linear algebra for embedded optimization (x). This second iteration explicitly targets SIMD instructions by means of intrinsics, while the matrices are still stored in column-wise format. The optimal `gemm` kernel size is now 8×4. Since the AVX instruction set can operate on 4 doubles at a time, the performance



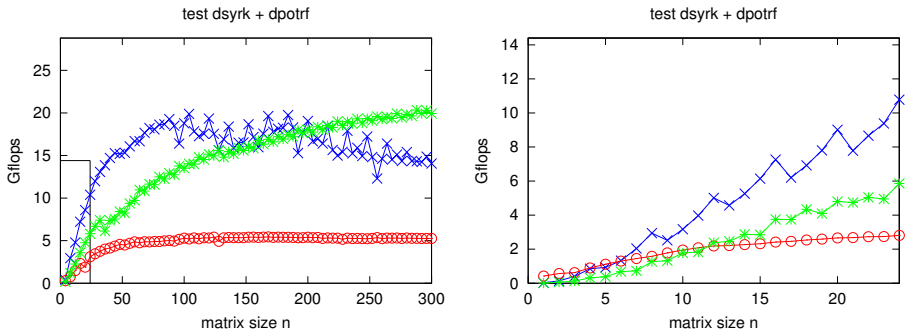
(a) (b)

Figure 3.5

shows a big improvement, arriving at a maximum of 20 Gflops (or 68% of full FP throughput) for a matrix size of $n = 168$. However, the performance is rather erratic, exacerbating the behavior already seen in the first iteration. For matrix size larger than about $n = 150$, OpenBLAS gives the best performance.

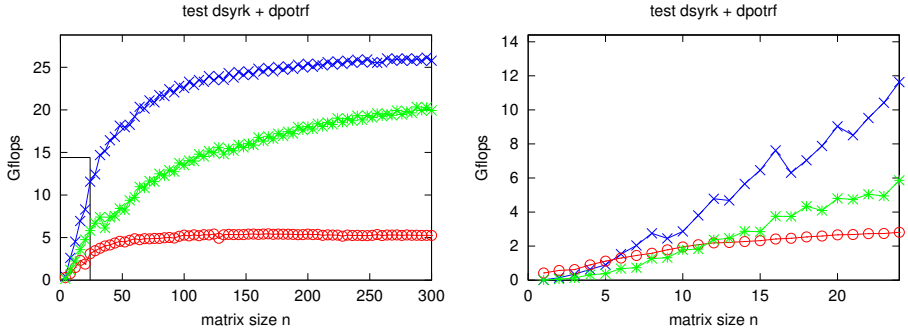
Figure 3.7 shows the performance plot of the third iteration of the proposed linear algebra for embedded optimization (x). This third iterations makes use of the panel-major matrix format. As a result, the performance is much more regular, since this matrix format improves cache usage, and therefore also for relatively large matrices data can be streamed from cache fast enough to keep execution units busy. In the large-scale test (Figure 3.7a), the performance keeps increasing with the matrix size, and the maximum performance is of 26 Gflops (90% of full FP throughput). For even larger matrices, the performance would decrease since block for cache is not employed; however, the performance is close to full FP throughput for the matrix sizes of interest in most embedded optimization applications. In the small-scale test (Figure 3.7b), the performance is almost identical to the second iteration case in Figure 3.6b: in fact, the panel-major matrix format gives no improvements if the matrices are small enough to entirely fit in L1 cache even in column-major format.

In the implementation of the `dsyrk` routine, the result matrix is build panel-wise, one block at at time: therefore in the innermost loop around the `gemm` kernel, two panels from the A matrix are reused, while the B matrix is streamed one panel at a time. On the contrary, in the implementation of the `dpotrf` routine, the result matrix is build across panels, one block at a time. This was the default choice in early implementations of the `dpotrf` routine in HPMPC, due



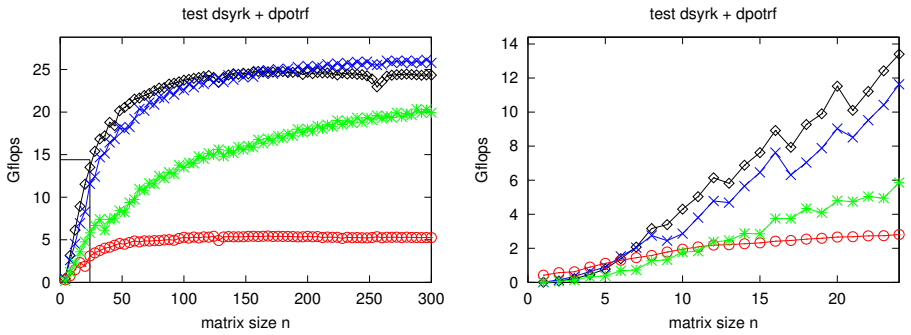
(a) (b)

Figure 3.6



(a) (b)

Figure 3.7



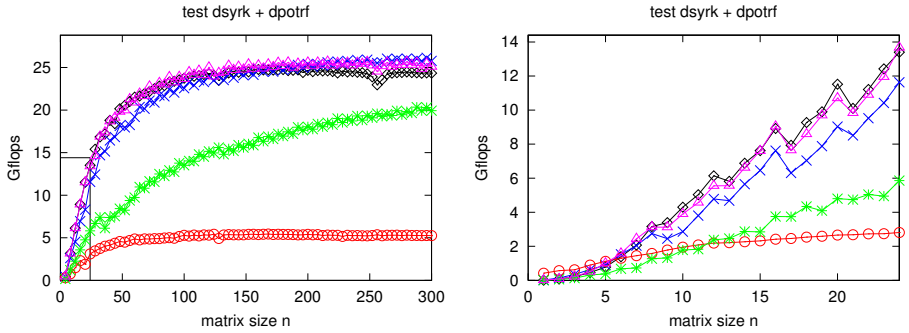
(a) (b)

Figure 3.8

to the possibility of storing in a favorable format and reusing the triangular matrix needed in the triangular substitutions. However, this means that in the innermost loop around the `gemm` kernel, only one panel from the A matrix is reused, while the B matrix needs to be streamed two panels at a time, requiring twice the memory bandwidth if the B matrix is not already in L1 cache.

Figure 3.8 shows the performance plot of the fourth iteration of the proposed linear algebra for embedded optimization ($\diamond$). This fourth iteration implements merging of linear algebra routines. Namely, the `dsyrk` and the `dpotrf` routines are merged into the single `dsyrk_dpotrf` routine. In Figure 3.8 it is chosen to keep the loop order of the `dpotrf` routine: therefore the result matrix is computed across panels. The performance for small matrices gets a good improvement compared to the use of un-merged routines, as shown in Figure 3.8b. However, the performance for large scale matrices gets slightly worse (3.8a). This is due to the fact that also the `dsyrk` part employed the sub-optimal outer loops order.

Figure 3.9 shows the performance plot of the fifth and last iteration of the proposed linear algebra for embedded optimization (Δ). In this case, the favorable matrix format for the triangular matrix used in the triangular substitution is dropped. This slightly lowers performance for small matrices (Figure 3.9b). However, this gives the possibility to implement `dsyrk_dpotrf` (and therefore also `dpotrf` alone) using the optimal outer loops order around the kernel. Therefore, the result matrix is build panel-wise, and only one panel from the B matrix of the `gemm` kernel needs to be streamed, while two panels from the A matrix



(a) (b)

Figure 3.9

of the `gemm` kernel are reused between iterations and therefore likely already present in L1 cache. This gives a performance for larger matrices (Figure 3.9a) only slightly lower than the un-merged routines.

The resulting routine therefore shows a good performance in both the small scale and the large scale tests. It outperforms reference BLAS and OpenBLAS for all matrix sizes of interest (for very large matrices, OpenBLAS performs better due to the use of blocking for cache). The code-generated version of the reference BLAS can outperform the proposed routine only for extremely tiny matrices, of size smaller than $n = 5$. This is due to the fact that the proposed routine is implemented in library format, and therefore a number of branches have to be taken in order to select the correct loops sizes. If extremely fast routines are needed in case of these extremely tiny matrices, the best option is to hand code these special cases using the proposed techniques, and hand remove all loops and branches.

3.5 Performance of level 3 BLAS and LAPACK routines

This section contains performance plots for the level 3 BLAS and LAPACK routines that constitute the backbone of the optimization algorithms presented in Part II and Part III of this thesis. For each routine, four versions are compared: the version implemented using the techniques presented in this thesis (HPMPC), the reference BLAS version (Netlib BLAS and LAPACK 3.5.0), an

optimized vendor BLAS (Intel's MKL 11.3) and an optimized open-source BLAS (OpenBLAS 0.2.15). Both MKL and OpenBLAS provide an optimized version of the tested LAPACK routines.

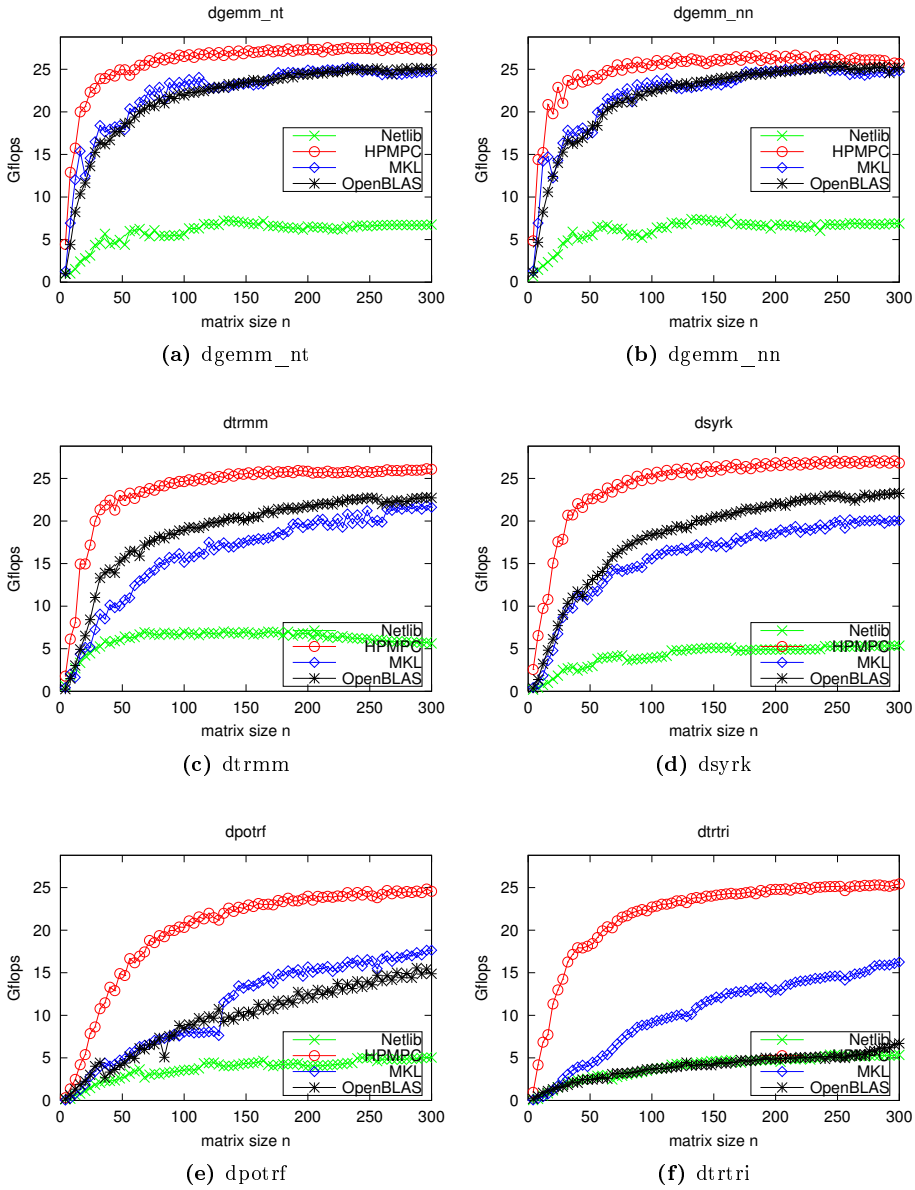
All routines are tested on two processors, implementing Intel Ivy-Bridge and Intel Haswell micro-architectures. The former supports the AVX ISA, while the latter supports the AVX2 and FMA ISAs, that are the best ISAs supported even in more recent micro-architectures such as Intel Broadwell and Intel Skylake. The two test machines have the same memory configuration, namely 8 GB of DDR3/DDR3L memory in dual-channel configuration (for a total data width of 128 bits), running at 1600 MHz, that gives a maximum bandwidth of 25.6 GB/s. Therefore, the difference in performance is solely due to the processors. These tests also show the current support state of the latest and previous latest x86_64 ISAs regarding FP.

The tests are performed in double precision and, unless differently stated, for squared matrices of size n between 4 and 300, in steps of 4 (and therefore they are meant to evaluate the performance for matrix size of interest for embedded optimization applications). In all tests, only one thread is employed: therefore, the single-thread version of optimized BLAS libraries is considered.

3.5.1 Performance on Intel Ivy-Bridge micro-architecture

The test processor is the same as in Section 3.4, namely an Intel Core i7 3520M running at the maximum turbo frequency of 3.6 GHz. The processor implements the Intel Ivy-Bridge micro-architecture, and it supports the AVX ISA. It can perform a 256-bit-wide FP multiplication and a 256-bit-wide FP addition every clock cycle, and therefore in double precision the full FP throughput for single-threaded code is of 8 flops per cycle (28.8 Gflops at 3.6 GHz). The AVX ISA has been employed in several micro-architectures, and is generally well supported in optimized BLAS version.

The performance plots are in Figure 3.10. The overall result of the tests is that the routines in HPMPC give (sometimes much) better performance for small matrices that both optimized BLAS versions. In the case of the `dgemm` routines, the difference in performance is relatively small. However, the performance gap increases for specialized BLAS routines (such as `dtrmm` and `dsyrk`), and even more for LAPACK routines (such as `dpotrf` and `dtrtri`). Generally speaking, OpenBLAS seems to have better performance than MKL on level 3 BLAS routines, but worse on LAPACK routines. The reference BLAS and LAPACK versions give poor performance on all tests.



(a) dgemm_nt **(b)** dgemm_nn

(c) dtrmm **(d)** dsyrk

(e) dpotrf **(f)** dtrtri

Figure 3.10: Performance test for key level 3 BLAS and LAPACK routines on an Intel core i7 3520M processor (Ivy Bridge micro-architecture, supporting the AVX ISA).

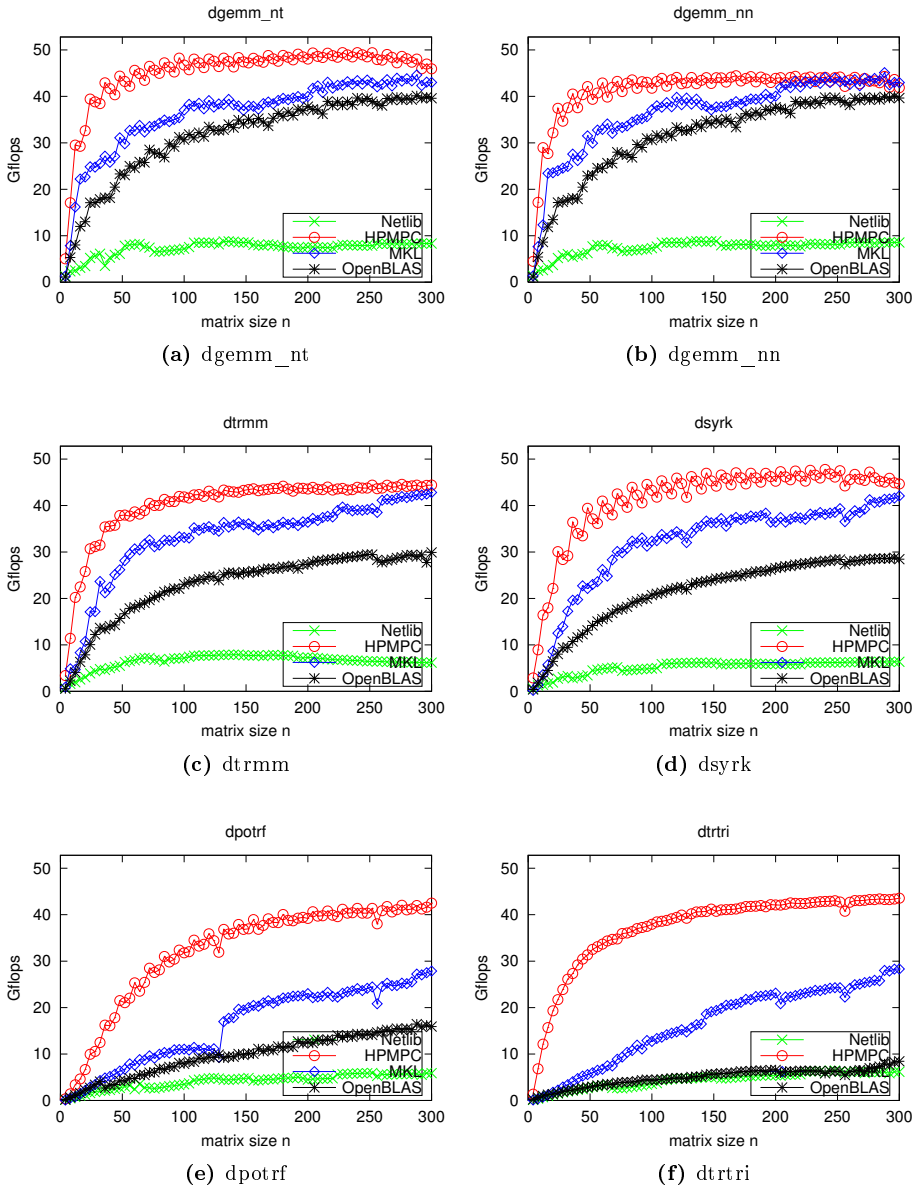
3.5.2 Performance on Intel Haswell micro-architecture

The test processor is an Intel Core i7 4800MQ. This processor has a base frequency of 2.7 GHz and a maximum turbo frequency of 3.7 GHz. However, if the 256-bit wide SIMD units are employed, the turbo frequency is lowered to 3.3 GHz, that is the frequency observed in these tests. The processor implements the Intel Haswell micro-architecture, and it supports the AVX2 and FMA ISA. It can perform 2 256-bit-wide FP fused-multiplication-addition every clock cycle, and therefore in double precision the full FP throughput for single-threaded code is of 16 flops per cycle (52.8 Gflops at 3.3 GHz). The AVX2 and FMA ISAs have been employed only in the most recent micro-architectures, and often they are not yet fully employed in optimized BLAS versions.

For the test on the Intel Haswell micro-architecture, a custom `gcc` compiler has been employed. See Appendix A for more details.

The performance plots are in Figure 3.11. Also in this case, the routines in HPMPC give better performance for small matrices. In general, MKL routines perform better that OpenBLAS ones on all tests, showing that these recent ISAs are not yet completely exploited in the open-source BLAS version. In particular, the `dtrmm` routine has been optimized for the AVX2 and FMA ISAs only in the latest OpenBLAS version (0.2.15), and the performance of the LAPACK routines is still very poor. On the contrary, the vendor MKL version give reasonably good performance on all tests.

Generally speaking, the routines in HPMPC give excellent performance. It is interesting to notice as for this micro-architecture there is a noticeable difference in performance between the optimal 'NT' version and the sub-optimal 'NN' version of the `dgemm` kernel. This is due to the fact that the Haswell doubles the full FP throughput with respect to the Ivy-Bridge micro-architecture, and therefore more care is needed in the data streaming from memory in order to keep the execution units busy. The performance advantages of HPMPC routines over optimized BLAS routines are particularly big for LAPACK routines. This is especially true for the `dtrtri` routine, where the huge difference in performance for small matrices is partially due to the fact that the custom routine in HPMPC can avoid to re-compute the reciprocal of the diagonal elements, if these have already been computed e.g. by a previous call to the `dpotrf` routine. The reference BLAS and LAPACK versions give very poor performance, that shows very little improvement with respect to the tests on the Ivy-Bridge micro-architecture: this shows that a more recent processor does not necessarily give better performance, if the code does not exploit the new features.



(a) dgemm_nt **(b)** dgemm_nn

(c) dtrmm **(d)** dsyrk

(e) dpotrf **(f)** dtrtri

Figure 3.11: Performance test for key level 3 BLAS and LAPACK routines on an Intel core i7 4800MQ processor (Haswell micro-architecture, supporting the AVX2 and FMA ISAs).

3.5.3 Performance in case of low rank updates

In this section, the performance of some linear algebra routine is tested in the case of low-rank updates. Namely, the 'NT' version of the `dgemm` routine and the `dsyrk` routine are tested for matrices with fixed $m = n = 100$ and variable and small $k \in [1, 24]$. Tailored routines to the low rank case are compared with the standard routines and to several BLAS implementations. The cases of rank equal to 1, 2, 3 and 4 are explicitly coded in the routines tailored to low rank updates; higher ranks are obtained looping over the rank 4 kernel. Results are in Figure 3.12.

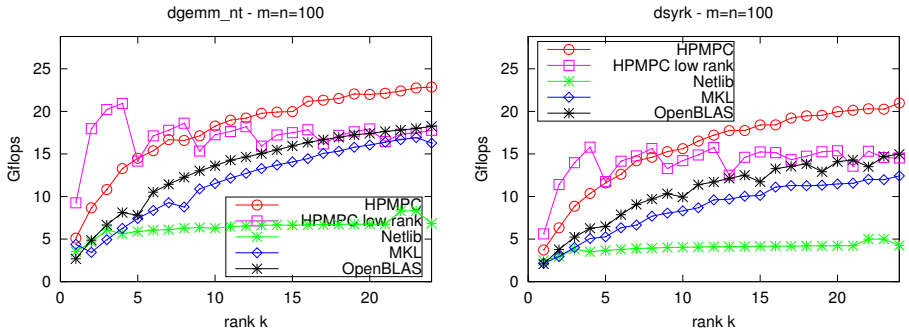
As a general figure, the performance improves rather regularly as the rank of the update increases. For very small ranks, the performance is particularly low. For ranks equal to 1, 2, 3 and 4 (i.e. the explicitly coded cases) the routines tailored to low rank updates show a great performance improvement. However, the performance does not improve for larger ranks, showing a clear period-4 pattern (since the rank 4 kernel has the highest performance). The explicit coding of higher ranks (that is possible as long as there are enough FP registers) would improve performance for these cases, at the expense of requiring a larger number of kernels. For the considered case, a safe choice is to prefer the low rank routines version for $k \leq 4$, and the standard routines version for $k > 4$.

Interestingly, for both the Ivy-Bridge and the Haswell architectures, the performance of the `dgemm` routine in MKL is higher for rank 1 than for rank 2, hinting at the fact that the rank 1 case is explicitly coded in MKL too. Furthermore, the reference Netlib implementation has better performance in case of very low ranks than optimized BLAS routines, due to the different order of the loops.

3.6 Conclusion

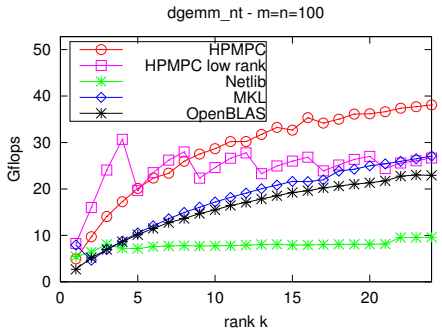
This section presented some level 3 BLAS linear algebra implementation techniques specially tailored for embedded optimization, and proposed a matrix format giving optimal performance for the proposed linear algebra routines, in case of matrices roughly fitting in LLC. Even if each single implementation technique is not new, their combination provides a novel implementation strategy for the linear algebra in embedded optimization.

In the case of level 3 BLAS and LAPACK routines, the bulk of the computation is cast in terms of the loop in the `gemm` kernel. LAPACK routines are implemented in the same fashion as level 3 BLAS routines, namely using tailored

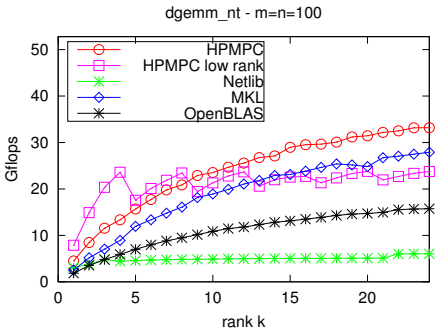


(a) dgemm_nt Ivy-Bridge

(b) dsyrk Ivy-Bridge



(c) dgemm_nt Haswell



(d) dsyrk Haswell

Figure 3.12: Performance test on low rank updates. The matrices size m and n are fixed to 100, while the rank update $k \in [1, 24]$.

kernels based on the optimal `gemm` kernel.

All steps in the optimization process are described in details, in a pedagogical fashion, and numerical tests show the performance advantages arising from each single step.

Numerous numerical tests confirm that the routines in HPMPC give a nice performance advantage over the corresponding level 3 BLAS and LAPACK routines in optimized BLAS libraries or code-generators for embedded optimization, for the matrix size of interest for embedded optimization applications.

As a final note, reference BLAS and LAPACK routines (that are the model of current code-generator for embedded optimization) perform poorly, and their performance shows very little improvement in the Haswell micro-architecture compared with the Ivy-Bridge micro-architecture, despite the fact that the former supports the AVX2 and FMA ISAs and has double the full FP throughput. This is a good example of the fact that a more recent processor does not necessarily give better performance, if the code does not exploit the new features.

Chapter 4

Level 2 BLAS for embedded optimization

Level 2 BLAS contains routines for basic matrix-vector operations. In the embedded optimization framework, level 2 BLAS routines are the backbone of first order optimization methods, and are used together with level 3 BLAS routines in the implementation of second order optimization methods.

If n is the size (number of rows or columns) of the matrix involved in a generic level 2 BLAS operation, then the computational cost of the operation is of about $\mathcal{O}(n^2)$ flops, that is the same as to the matrix size in memory. Therefore, each matrix element is reused $\mathcal{O}(1)$ times (and often exactly once) in a level 2 BLAS operation. In modern architectures the time needed to move a matrix element from memory into registers is much larger than the time needed to perform a floating-point operation on that element once in registers. Therefore, the cost of level 2 BLAS routines is typically dominated by the cost of streaming the matrix from memory into registers. As a consequence, these routines can typically attain a low fraction of the full FP throughput, and the performance decreases as the matrix size increases, exceeding cache size. This is a key difference with respect to the level 3 BLAS and LAPACK routines.

For large matrices, the computational cost of level 2 BLAS routines is negligible with respect to the cost of level 3 BLAS and LAPACK routines. However, for the

small matrix sizes typical of embedded optimization the cost of level 2 BLAS routines is often not negligible. Furthermore, in the implementation of first order optimization methods, there are not level 3 BLAS and LAPACK routines. Therefore, in the framework of embedded optimization, the performance of level 2 BLAS routines plays an important role too.

In the implementation of level 2 BLAS routines, most of the computation can be cast in term of two routines: the `gemv` routine (that is the general matrix-vector multiplication routine), and to a smaller extent the `symv` routine (that is the symmetric matrix-vector multiplication routine). The role of the `gemv` routine is analogue to the role of the `gemm` routine in level 3 BLAS and LAPACK routines, as most of level 2 BLAS routines can be implemented using a kernel for `gemv`. A notable exception is the `symv` routine: even if it is possible to implement it by means of two triangular matrix-vector multiplications (and therefore making use of the `gemv` kernel), since it has a reuse factor of 2 it makes sense to write a specialized kernel exploiting this to reduce bandwidth requirements. Furthermore, in optimization symmetric matrices are relatively widespread (e.g. the Hessian of the cost function in optimization problems), therefore many applications can benefit from a specialized `symv` kernel.

The assumptions in Section 3.1 about the nature of the embedded optimization problems still hold. In particular, level 2 BLAS routines are designed with special care on reducing overhead and enhancing small scale performance. All matrices are assumed to be stored in the panel-major format proposed in Section 3.2.2. This is the optimal matrix format for the 'NT' version of `gemm` optimized for small matrices. It provides advantages over row-major or column-major matrix orders also in the case of the `gemv` and `symv` kernels. In the case of level 2 BLAS routines, since there is $\mathcal{O}(1)$ reuse of matrix elements, the explicit packing or transposition of matrices is never advantageous, and therefore it is not employed in optimized BLAS libraries, that assume row-major or column-major matrix formats for level 2 BLAS routines. In the proposed implementation approach for embedded optimization, however, the conversion of matrices into the panel-major format is assumed to be performed off-line or to be well amortized e.g. over the several iterations of an optimization method. Therefore, the use of the panel-major matrix format can enhance performance with respect to level 2 BLAS routines in optimized BLAS libraries.

In the proposed linear algebra routines for embedded optimization, vectors are assumed to be stored contiguously in memory (this is equivalent to `incx` and `incy` equal to 1 in standard level 2 BLAS routines). This implies that it is not possible to use the proposed level 2 BLAS routines to operate on a single row or column of matrices stored in the panel-major matrix format. This simplifies the implementation of level 2 BLAS routines, at the cost of reducing flexibility. However, this has not been found to be an issue in the implementation of op-

timization algorithms. In the rare cases where operations with matrix rows or columns are necessary, explicit copy into a contiguous vector is employed. This can also be seen as the packing of a vector into a favorable format, since each vector element is reused $\mathcal{O}(n)$ times in level 2 BLAS routines, and unit stride optimizes the reuse of cache lines and allows vectorization.

This chapter is organized as follows. Section 4.1 and 4.2 presents in details the optimization of the key routine in level 3 BLAS: the `gemv` and `symv` respectively. Section 4.3 presents the optimization of other level 2 BLAS routines. Finally, Section 4.4 contains the performance plot for some widely used level 2 BLAS routines.

4.1 Optimizing the gemv routine

The `gemv` routine is the general matrix-vector multiplication routine. In the BLAS standard, it has the interface (considering the double precision version, and using C notation)

```
dgemv_(char *trans, int *m, int *n, double *alpha, double *A, \
    int *lda, double *x, int *incx, double *beta, double *y, \
    int *incy);
```

and it computes

$$y \leftarrow \alpha \cdot \mathrm{op}(A) \cdot x + \beta \cdot y$$

where α and β are scalars, and $\mathrm{op}(A)$ can be either A of A^T depending on the flag `trans`. The matrix A has size $m \times n$. The matrix A is stored in column-major (of Fortran-like) order, where consecutive elements on the same row are stored `lda` elements away. The elements in the vectors x and y are stored `incx` and `incy` elements away. This can be used to operate on single rows or columns of matrices, giving great flexibility.

The following alternative interfaces are considered in this thesis:

```
void dgemv_n_lib(int m, int n, double *A, int sda, \
    double *x, int alg, double *y, double *z);
```

computing

$$z \leftarrow \alpha \cdot A \cdot x + \beta \cdot y$$

and

```
void dgemv_t_lib(int m, int n, double *A, int sda, \
    double *x, int alg, double *y, double *z);
```

computing

$$z \leftarrow \alpha \cdot A^T \cdot x + \beta \cdot y$$

where α and β are controlled by the flag `alg` in the same way as in the `gemm` routine, as

$$\texttt{alg} = \begin{cases} 0 & \Rightarrow \quad \alpha \leftarrow 1, \beta \leftarrow 0 \\ 1 & \Rightarrow \quad \alpha \leftarrow 1, \beta \leftarrow 1 \\ -1 & \Rightarrow \quad \alpha \leftarrow -1, \beta \leftarrow 1 \end{cases}$$

In both versions, the matrix A has size $m \times n$.

Notice that these routines take 1 matrix operand and 3 vector operands, meaning that the result vector z does not necessarily overwrite the vector y: it can do so if y and z correspond to the same memory location. This feature is useful in many cases to avoid an explicit vector copy.

Compared to the standard `gemv` interface, these alternative formulation are somehow lower-level interfaces, closer to the interface of the underlying `gemv` kernels. This has the advantage of reducing the overhead of the routine, increasing performance for small matrices. As a drawback, they are less general, covering just the cases commonly encountered in embedded optimization.

4.1.1 Optimizing the `gemv` kernel

The `gemv` kernels are the routines accounting for the innermost loop in the implementation of the different `gemv` variants. These kernels compute a fixed-size sub-vector of the result vector z by adding a fixed-size vector of y with the product of a panel from A and the vector x, where one of the two panel dimensions are fixed. Each iteration of the kernel loop computes a rank-1 update of the result sub-vector. In order to achieve the best performance, the `gemv` kernels are optimized for different computer architectures. Therefore features as e.g. the height of the panel and the size of the fixed-size sub-vector of z computed by the kernels are architecture-dependent. As an example, the 'N' variant of the `gemv` kernel computing a 8×1 sub-vector of z, where the A matrix is assumed to be stored in the panel-major format with panel height $b_s = 4$ (see Section 3.2.2 for more details), is

```
void kernel_dgemv_n_8_lib4(int kmax, double *A, int sda, \
    double *x, int alg, double *y, double *z);
```

Notice that the interface of the gemm routines closely resemble the interface of the corresponding underlying gemm kernel.

The gemv kernels are the key kernels in all level 2 BLAS routines, since the computationally most expensive parts of all level 2 BLAS routines can be cast in term of these kernels. This section presents the generic techniques used in the implementation of all gemv kernels, and shows how to optimize them for an hypothetical computer architecture.

4.1.1.1 Blocking for registers

In the gemv routine, and more generally in all level 2 BLAS routines, the reuse factor of matrix elements is $\mathcal{O}(1)$ (and usually exactly 1, with the exception of the symv routine, where it is 2). Therefore the gemv routine is generally memory-bound, meaning that the factor limiting the attained performance is the memory bandwidth and not the computational throughput. Well-optimized level 2 BLAS routines can attain a good performance if the matrix data is already in L1 cache, and the performance decreases as the data needs to be loaded from lower levels of cache or from main memory.

Blocking for registers can be employed also in the implementation of gemv, where it can provide enough independent instructions to hide instruction latency, and to a limited extent it can reduce memory bandwidth. In fact, matrix elements have a reuse factor of $\mathcal{O}(1)$, but vector elements have a reuse factor of $\mathcal{O}(n)$, and therefore the memory bandwidth requirements can be almost halved.

The same hypothetical processor introduced in Section 3.2.1.1 is considered in the optimization choices. This processor can perform one FMA (fused-multiply-add) every clock cycle (throughput=1), while the result is available for further computations after 4 cycles (latency=4). Furthermore, this processor can load one floating point register from L1 cache every clock cycle.

In the case of the gemv routine, the kernels for the 'N' and for the 'T' versions are analogue when only scalar instructions are employed, but they are fundamentally different when vector instructions are employed.

In the 'N' version case, without loss of generality the following gemv operation is considered: a squared matrix A of size $n \times n$ is multiplied by a vector x of size n, and the result is used to update the content of a vector y of size n, as

$$y \leftarrow y + A \cdot x.$$

Using the definition of matrix-vector product, each element y_i of y can be computed as the inner product

$$y_i = y_i + \sum_{k=0}^{n} a_{ik} \cdot x_k$$

that is performed using the sequence of n FMA instructions

$$\begin{array}{c|c} & x_k \\ \hline a_{ik} & y_i \leftarrow y_i + a_{ik} \cdot x_k \end{array} \quad , \quad k = 0, 1, \ldots, n-1. \tag{4.1}$$

The dependency pattern is analogue to the one in (3.1): each FMA depends on the result of the previous FMA. If the data is present in L1 cache, a FMA can be performed every 4 clock cycles. In any case, 2 FP numbers (a_{ik} and x_k) need to be loaded from memory to perform each FMA. Therefore also ignoring for a moment the latency constraint, it would not be possible to perform 1 FMA every clock cycle and keep the pipeline full due to the memory bandwidth constraint from L1 cache.

Block for cache can mitigate these latency and bandwidth constraints also in the case of the `gemv` kernel. If 4 FP registers are used to hold a 4×1 sub-vector of y, the 4 elements of the sub-vector can be computed simultaneously (using 0, 1, 2, 3 in place of $i+0$, $i+1$, $i+2$, $i+3$ to keep the notation lighter) as

$$\begin{array}{c|c} & x_k \\ \hline a_{0k} & y_0 \leftarrow y_0 + a_{0k} \cdot x_k \\ a_{1k} & y_1 \leftarrow y_1 + a_{1k} \cdot x_k \\ a_{2k} & y_2 \leftarrow y_2 + a_{2k} \cdot x_k \\ a_{3k} & y_3 \leftarrow y_3 + a_{3k} \cdot x_k \end{array} \quad , \quad k = 0, 1, \ldots, n-1. \tag{4.2}$$

Again, the dependency pattern is analogue to the one in (3.2), with a FMA instruction depending on the result of the 4th-last FMA instruction. Regarding the instruction latency issue, the use of 4 accumulation registers can totally hide FMA instruction latency. Regarding the memory bandwidth issue, even assuming that required data is present in L1 cache, the `gemv` kernel is memory-bound. In fact, each x element is now reused 4 times once in registers, but the A elements are not, and therefore 5 FP loads needs to be performed every 4 cycles (4 elements from A and 1 element from x), while the processor can perform at most 4 FP loads. Therefore, even assuming that the data is already present in L1 cache, this `gemv` kernel can attain at most 80% of the scalar FP throughput of the processor.

In practice, if the required data is not already present in L1 cache, the performance is further limited by the main memory bandwidth, that is generally not high enough to stream the matrix fast enough and keep execution units busy.

A similar analysis can be done for the 'T' version of the gemv kernel. Let us consider without loss of generality the operation

$$y \leftarrow y + A^T \cdot x.$$

Using the definition of matrix-vector product when the matrix is transposed, each element y_i of y can be computed as the inner product

$$y_i = y_i + \sum_{k=0}^{n-1} a_{ki} \cdot x_k$$

where the only difference with respect to the 'N' version is the fact that the indexes of the A elements are swapped. The equivalent of (4.2) for the 'T' variant is

	a_{k0}	a_{k1}	a_{k2}	a_{k3}
x_k	$y_0 \Leftarrow a_{k0} \cdot x_k$	$y_1 \Leftarrow a_{k1} \cdot x_k$	$y_2 \Leftarrow a_{k2} \cdot x_k$	$y_3 \Leftarrow a_{k3} \cdot x_k$

(4.3)

(where the simbol $\Leftarrow$ is used as a short expression for the accumulation operator, and therefore $y_0 \Leftarrow a_{k0} \cdot x_{ka}$ is equivalent to $y_0 \leftarrow y_0 + a_{k0} \cdot x_k$) for $k = 0, 1, \ldots, n-1$. The analysis is analogue to the 'N' case.

4.1.1.2 Use of SIMD instructions

If available on the processor, SIMD (Single-Instruction Multiple-Data, that are instructions that perform the same operation in parallel on all elements of a small vector of data) can be used to improve the performance of the gemv kernel in case the limiting factor is not the memory bandwidth from main memory.

Continuing the example, let us assume that the hypothetical processor has 2-wide vector units (i.e. each holding 2 FP numbers), and vector FMA and load instructions (operating on the 2-wide vectors) with the same latency and throughput of the scalar instructions used in Section 4.1.1.1.

The 'N' kernel operating on the 4×1 sub-vector in (4.2) can be implemented using vector instructions as

	x_k
a_{0k} a_{1k}	$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} \leftarrow \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} + \begin{bmatrix} a_{0k} \\ a_{1k} \end{bmatrix} \cdot \begin{bmatrix} x_k \\ x_k \end{bmatrix}$
a_{2k} a_{3k}	$\begin{bmatrix} y_2 \\ y_3 \end{bmatrix} \leftarrow \begin{bmatrix} y_2 \\ y_3 \end{bmatrix} + \begin{bmatrix} a_{2k} \\ a_{3k} \end{bmatrix} \cdot \begin{bmatrix} x_k \\ x_k \end{bmatrix}$

$$, \quad k = 0, 1, \ldots, n-1.$$

where the squared brackets indicates the small vectors. Also in the gemv case, as a result the number of instructions is halved. Therefore there are not enough

independent instructions, and the attained throughput is only slightly larger than in the scalar case: the vector kernel attains the 50% of the vector throughput, while the scalar kernel attains the 40% (since 80% of scalar throughput is equivalent to 40% of 2-wide-vector throughput).

In order to increase the number of independent instructions, it is possible to compute an additional 4×1 sub-vector of y. Therefore, the `gemv` kernel computes a 8×1 sub-matrix of y as

$$\begin{array}{c|c} & x_k \\ \hline a_{0k} & \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} \leftarrow \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} + \begin{bmatrix} a_{0k} \\ a_{1k} \end{bmatrix} \cdot \begin{bmatrix} x_k \\ x_k \end{bmatrix} \\ a_{1k} & \\ a_{2k} & \begin{bmatrix} y_2 \\ y_3 \end{bmatrix} \leftarrow \begin{bmatrix} y_2 \\ y_3 \end{bmatrix} + \begin{bmatrix} a_{2k} \\ a_{3k} \end{bmatrix} \cdot \begin{bmatrix} x_k \\ x_k \end{bmatrix} \\ a_{3k} & \\ a_{4k} & \begin{bmatrix} y_4 \\ y_5 \end{bmatrix} \leftarrow \begin{bmatrix} y_4 \\ y_5 \end{bmatrix} + \begin{bmatrix} a_{4k} \\ a_{5k} \end{bmatrix} \cdot \begin{bmatrix} x_k \\ x_k \end{bmatrix} \\ a_{5k} & \\ a_{6k} & \begin{bmatrix} y_6 \\ y_7 \end{bmatrix} \leftarrow \begin{bmatrix} y_6 \\ y_7 \end{bmatrix} + \begin{bmatrix} a_{6k} \\ a_{7k} \end{bmatrix} \cdot \begin{bmatrix} x_k \\ x_k \end{bmatrix} \\ a_{7k} & \end{array} \quad , \quad k = 0, 1, \ldots, n-1.$$

Notice that this time each x element is reused 8 times. This kernel has enough independent FMA instructions to totally hide FMA instruction latency. However, even if the data is already present in L1 cache, the L1 cache bandwidth limits the maximum performance. In fact, depending on the ISA, this kernel can attain at most 80.0% of the full FP throughput (if the x_k element is loaded and broadcast to all vector components with a single instruction), or 88.9% (if the elements x_k and x_{k+1} for two consecutive loop iterations are loaded every two loop iterations with a single load instruction). Notice that an entire column of 8 elements from A needs to be processed at the same time in order to obtain a well-optimized kernel. In case of wider vector units or longer FMA latency, this number will be even larger. Therefore, in case of small matrices, this affects the performance of the `gemv` kernel more than the performance of the `gemm` kernel.

In the vector version of the 'T' `gemv` kernel, vectorization is used to compute multiple loop iterations with a single instruction, instead of to compute several y elements with a single instruction as in the 'N' `gemv` kernel. In order to make the notation easier, the elements of the first iteration are denoted with $_h$ and the elements of the second iteration with $_k$. The 'T' `gemv` kernel operating on the 4×1 sub-vector of y computes two following loop iterations and it can be implemented using vector instructions as

$$\begin{array}{c|cccc} & \begin{bmatrix} a_{h0} \\ a_{k0} \end{bmatrix} & \begin{bmatrix} a_{h1} \\ a_{k1} \end{bmatrix} & \begin{bmatrix} a_{h2} \\ a_{k2} \end{bmatrix} & \begin{bmatrix} a_{h3} \\ a_{k3} \end{bmatrix} \\ \hline \begin{bmatrix} x_h \\ x_k \end{bmatrix} & \begin{bmatrix} y_0^0 \\ y_0^1 \end{bmatrix} \Leftarrow \begin{bmatrix} a_{h0} \\ a_{k0} \end{bmatrix} \cdot \begin{bmatrix} x_h \\ x_k \end{bmatrix} & \begin{bmatrix} y_1^0 \\ y_1^1 \end{bmatrix} \Leftarrow \begin{bmatrix} a_{h1} \\ a_{k1} \end{bmatrix} \cdot \begin{bmatrix} x_h \\ x_k \end{bmatrix} & \begin{bmatrix} y_2^0 \\ y_2^1 \end{bmatrix} \Leftarrow \begin{bmatrix} a_{h2} \\ a_{k2} \end{bmatrix} \cdot \begin{bmatrix} x_h \\ x_k \end{bmatrix} & \begin{bmatrix} y_3^0 \\ y_3^1 \end{bmatrix} \Leftarrow \begin{bmatrix} a_{h3} \\ a_{k3} \end{bmatrix} \cdot \begin{bmatrix} x_h \\ x_k \end{bmatrix} \end{array}.$$

(where $\Leftarrow$ is a short for the accumulation operator, as in (4.3)) for $h = 0, 2, \ldots, n-$

2 and $k = 1, 3, \ldots, n-1$ (assuming n even). The exponents ^{0} and ^{1} are used to represent the two partial sums of each y_i element. At the end of the main loop, the partial sums in the registers need to be reduced (i.e. all components of a single vector have to be summed together to compute the result element y_i). Depending on the ISA, this can be a rather expensive operation, but it is amortized over $^n/_2$ effective loop iterations. Furthermore, at the end of the main `gemv` loop, a scalar clean-up loop iteration is required in case n is odd. Due to the L1 cache bandwidth, this kernel can attain at most the 80% of the full FP throughput. However, in practice the performance is reduced by the vectors reduction overhead. Furthermore notice that, in case of wider vectors of size n_v, even more loop iterations are computed at once, implying that the reduction overhead is amortized over an effective smaller number of iteration $^n/_{n_v}$.

Generally speaking, the use of SIMD can improve the effective memory bandwidth from L1 cache (if an entire vector can be loaded with a single instruction), but it does not affect the memory bandwidth from main memory. Therefore, in general the use of SIMD can improve the performance of the `gemv` kernel when the data is already present in cache, but not if data has to be loaded from main memory.

4.1.2 Use of contiguous memory and panel-major matrix format

In the implementation of the `gemv` kernel (and more in general of level 2 BLAS routines) the use of contiguous memory plays a less crucial role than in the implementation of the `gemm` kernel (and more in general of level 3 BLAS and LAPACK routines). This is due to the fact that in level 2 BLAS routines there is a reuse factor of matrix elements of $\mathcal{O}(1)$ (and generally exactly 1). Unless the matrix A is reused in immediately subsequent linear algebra routines without being evicted from cache, the use of a matrix format that optimizes cache reuse is of limited benefit.

Nevertheless, this matrix format has still some useful feature. The matrix panels are aligned to cache boundaries, and therefore once a cache line is moved from main memory to cache, it is fully utilized by both variants of the `gemv` kernel. Furthermore, the memory access pattern has some regularity. The regular access pattern of the 'N' version of the `gemv` kernel (i.e. the A matrix is read along panels) can be easily detected by the hardware prefetcher (if present in the architecture). The less regular access pattern of the 'T' version of the `gemv` kernel (i.e. the A matrix is read across panels, with only a few column from a panel accessed before moving to the following panel) can generally not be detected by the hardware prefetcher, but nevertheless it can be efficiently exploited by

means of software prefetch.

As a result, even if of limited benefit on its own, the panel-major matrix format provides a regular enough access pattern of matrix elements. This access pattern can be directly detected by the hardware prefetcher ('N' version of `gemv`), or explicitly exploited by means of software prefetch ('T' version of `gemv`), and this may help in moving data into cache before it is needed, enhancing performance.

4.1.3 Edges handling

In the `gemv` case, edges are handled similarly to what done in Section 3.2.4 for the `gemm` case. The main difference is that the `gemv` routines are implemented as a single loop around the `gemv` kernels. Therefore, the edges in one matrix dimensions are already handled by the loop within the kernels. On the other matrix dimension, a set of kernels of fixed and variable-store-size is implemented: this set has to handle edges size between 1 and the optimal kernel size in only one matrix dimension, greatly reducing the number of needed kernels.

One optimal `gemv` kernel for each of the 'N' and 'T' versions takes care of most of the computation load in case of large matrices. These kernels can store vectors of fixed size, and they do not have the overhead of the variable-store-size logic.

A set of variable-store-size kernels takes care of the edges for each of the 'N' and 'T' `gemv` versions. In case of the 'N' version, the kernel size is multiple of the SIMD width, since it does not make any difference in computational time if the vectors are fully utilized or not. In case of the 'T' version, it is possible to take advantage from a finer granularity, since the kernel size affects the number of instructions in the kernel loop, and not the degree of utilization of vectors as in the 'N' case.

4.2 Optimizing the `symv` routine

The `symv` routine is the symmetric matrix-vector routine. In the BLAS standard, it has the interface (considering the double precision version, and using C notation)

```
dsymv_(char *uplo, int *n, double *alpha, double *A, int *lda, \
      double *x, int *incx, double *beta, double *y, int *incy);
```

and it computes

$$y \leftarrow \alpha \cdot A \cdot x + \beta \cdot y$$

where α and β are scalars. The matrix A has size $n \times n$ and it is symmetric; the flag `uplo` controls if either the lower or the upper triangular part of A is accessed during computation. The matrix A is stored in column-major (of Fortran-like) order, where consecutive elements on the same row are stored `lda` elements away. The elements in the vectors x and y are stored `incx` and `incy` elements away. This can be used to operate on single rows or columns of matrices, giving great flexibility.

The following alternative interface is considered in this thesis:

```
void dsymv_lib(int m, int n, double *A, int sda, \
    double *x, int alg, double *y, double *z);
```

computing

$$z \leftarrow \alpha \cdot A \cdot x + \beta \cdot y$$

where α and β are controlled by the flag `alg` in the same way as in the `gemm` routine, as

$$\texttt{alg} = \begin{cases} 0 & \Rightarrow \quad \alpha \leftarrow 1, \beta \leftarrow 0 \\ 1 & \Rightarrow \quad \alpha \leftarrow 1, \beta \leftarrow 1 \\ -1 & \Rightarrow \quad \alpha \leftarrow -1, \beta \leftarrow 1 \end{cases}$$

The matrix A has size $m \times m$, where $m \geq n$. If $m = n$, A if a dense symmetric matrix. If $m > n$, A is a symmetric matrix in the form

$$A = \begin{bmatrix} A_0 & A_1^T \\ A_1 & \end{bmatrix}$$

where A_0 is a dense symmetric $n \times n$ matrix and A_1 is a dense generic $(m-n) \times n$ matrix. Matrices in this form are common in optimization (e.g. the KKT matrix of an equality constrained QP), and many applications can benefit from the the possibility of computing also the matrix-vector multiplications involving A_1 by means of a routine with reuse factor 2.

4.2.1 Optimizing the symv kernel

The `symv` kernel is the routine accounting for the innermost loop in the implementation of the `symv` routine. This kernel combines the 'N' and the 'T' versions of the `gemv` kernel, computing simultaneously

$$z_n \leftarrow \alpha \cdot A \cdot x_n + \beta \cdot y_n \quad \text{and} \quad z_t \leftarrow \alpha \cdot A^T \cdot y_t + \beta \cdot y_t$$

where A is a matrix panel, such that each A element is reused twice once in registers. In Section 4.1.1.2, it is shown that the vector implementation of the 'T' version of the `gemv` kernel access the matrix across panels, since this allows to amortize the vector reduction over n loop iterations. On the contrary, accessing the matrix along panels would imply that the vector reduction has to be computed at each loop iteration, greatly affecting performance. Therefore, also the `symv` kernel is required to access the matrix across panels: the 'T' part of the `symv` kernel is thus identical to the 'T' version of the `gemv` kernel, while the 'N' part has a different matrix access pattern with respect to the 'N' version of the `gemv` kernel. As a consequence, the `symv` kernel computes a fixed-size sub-vector of the result vector z_t, and a low-rank update of the result vector z_n. Each iteration of the kernel loop computes a rank-1 update of the sub-vector of the result vector z_t, and a low-rank update of an element of the result vector z_n.

In order to achieve the best performance, the `symv` kernel is optimized for different computer architectures. Therefore features as e.g. the height of the panel, the degree of the low-rank update of the result vector z_n and the size of the fixed-size sub-vector of the result vector z_t computed by the kernel are architecture-dependent. As an example, the `symv` kernel computing a rank-4 update of the result vector z_n and a 4×1 sub-vector of the result vector z_t, where the A matrix is assumed to be stored in the panel-major format with panel height $b_s = 4$ (see Section 3.2.2 for more details), is

```
void kernel_dsymv_4_lib4(int kmax, int tri_A, double *A, int sda, \
    double *x_n, double *x_t, int alg, double *y_n, double *y_t, \
    double *z_n, double *z_t);
```

Notice that the interface of the `symm` routines does not closely resemble the interface of the corresponding underlying `symm` kernel, as the latter has many more arguments. The flag `tri_A` is used to indicate whether the beginning of the A panel is a triangle or not.

The `symv` kernel can be used to compute the `symv` routine, and more generally to compute simultaneously the 'N' and 'T' versions of `gemv`. The routine computing this latter operation will be called `gemv_nt`, since it simultaneously accounts for the 'N' and 'T' versions of `gemv`. This section presents the generic techniques used in the implementation of all `symv` kernels, and shows how to optimize them for an hypothetical computer architecture.

4.2.1.1 Blocking for registers

In the symv, the reuse factor of the matrix elements is 2: this compares favorably with the reuse factor of the other level 2 BLAS routines, that is equal to 1. Therefore the symv routine is still expected to be memory-bound, but since the memory bandwidth requirements are lower, the performance is expected to be better than the gemv routine when the memory has to be loaded from main memory. On the other hand, the fact that the 'N' part of the symv kernel computes a low-rank update of the entire result vector z_n can introduce some issue regarding the possibility to hide instruction latency.

Blocking for registers can be employed also in the implementation of symv, where it can provide enough independent instructions to hide instruction latency, and to a limited extent it can reduce memory bandwidth.

The same hypothetical processor introduced in Sections 3.2.1.1 and 4.1.1.1 is considered in the optimization choices. This processor can perform one FMA (fused-multiply-add) every clock cycle (throughput=1), while the result is available for further computations after 4 cycles (latency=4). Furthermore, this processor can load one floating point register from L1 cache every clock cycle.

Let us consider the 'T' version of the gemv kernel computing a sub-vector of y_t of size 4 as basis for the symv kernel. This gemv kernel uses 4 registers to hold the 4 elements of the sub-vector of y_t, that are loaded before the main loop. At each loop iteration, one element from x_t and 4 elements from A are loaded and multiplied by means of 4 FMAs.

The symv kernel expands the 'T' version of the gemv kernel by using additional 4 registers to hold the 4 elements fo the sub-vector of x_n. At each loop iterations, one additional element from y_n is loaded, and 4 additional FMAs are performed. The resulting 8 FMAs computed at each loop iteration are

	a_{k0}	a_{k1}	a_{k2}	a_{k3}
y_k^n	$y_k^n \Leftarrow a_{k0} \cdot x_0^n$	$y_k^n \Leftarrow a_{k1} \cdot x_1^n$	$y_k^n \Leftarrow a_{k2} \cdot x_2^n$	$y_k^n \Leftarrow a_{k3} \cdot x_3^n$
x_k^t	$y_0^t \Leftarrow a_{k0} \cdot x_k^t$	$y_1^t \Leftarrow a_{k1} \cdot x_k^t$	$y_2^t \Leftarrow a_{k2} \cdot x_k^t$	$y_3^t \Leftarrow a_{k3} \cdot x_k^t$

(4.4)

(where $\Leftarrow$ is a short for the accumulation operator, as in (4.3)) for $k = 0, 1, \ldots, n-1$. It is immediately evident that the 4 FMAs coming from the 'T' part are independent, but the 4 FMAs coming from the 'N' part are not. Even issuing the FMAs from the 'T' part between the dependent FMAs from the 'N' part, there are not enough independent instructions to keep the FMA pipeline busy, as on average it is possible to issue a FMA every 2 cycles, for a maximum performance of 50% of the scalar full FP throughput.

It is possible to have enough independent FMA instructions by considering two consecutive iteration loops. In order to make the notation easier, the elements of the first iteration are denoted with $_h$ and the elements of the second iteration with $_k$. The resulting 16 FMAs computed in two consecutive loops are

$$\begin{array}{c|cccc} & a_{k0} & a_{k1} & a_{k2} & a_{k3} \\ \hline y_h^n & y_h^n \Leftarrow a_{h0} \cdot x_0^n & y_h^n \Leftarrow a_{h1} \cdot x_1^n & y_h^n \Leftarrow a_{h2} \cdot x_2^n & y_h^n \Leftarrow a_{h3} \cdot x_3^n \\ x_h^t & y_0^t \Leftarrow a_{h0} \cdot x_h^t & y_1^t \Leftarrow a_{h1} \cdot x_h^t & y_2^t \Leftarrow a_{h2} \cdot x_h^t & y_3^t \Leftarrow a_{h3} \cdot x_h^t \\ y_k^n & y_k^n \Leftarrow a_{k0} \cdot x_0^n & y_k^n \Leftarrow a_{k1} \cdot x_1^n & y_k^n \Leftarrow a_{k2} \cdot x_2^n & y_k^n \Leftarrow a_{k3} \cdot x_3^n \\ x_k^t & y_0^t \Leftarrow a_{k0} \cdot x_k^t & y_1^t \Leftarrow a_{k1} \cdot x_k^t & y_2^t \Leftarrow a_{k2} \cdot x_k^t & y_3^t \Leftarrow a_{k3} \cdot x_k^t \end{array} \tag{4.5}$$

Let us assume that the couples of FMAs involving the same A element are issued in sequence. This means that in the above scheme there are 8 couples of FMAs (2 rows of 4 columns each), and that full throughput can be achieved if each couple is followed by an independent couple. Each couple depends on all couples on the same row (due to the dependency between y_k^n elements), and each couple depends on the couple immediately above or below (due to the dependency between y_i^t elements). A possible sequence of couples that can be issued without stalling the FMA pipeline due to dependencies is

$$\begin{bmatrix} 1 & 5 & 3 & 7 \\ 6 & 2 & 8 & 4 \end{bmatrix} \tag{4.6}$$

where the number indicates the order of issue of the couple in the corresponding position.

During the 16 clock cycles needed to issue all the 16 independent FMAs, 8 elements are loaded from memory. Therefore, if the matrix and vector elements are in L1 cache, it is theoretically possible to achieve the scalar full FP throughput.

Notice that this scheme requires a considerable amount of registers: 4 to hold the x_i^n elements, 4 to hold the y_i^t elements, and 4 to hold y_h^n, y_k^n, x_j^t, x_k^t, plus at least 1 for the A elements and possibly 1 for the intermediate results (depending on the ISA), for a total of 13 or 14 FP registers.

4.2.1.2 Use of SIMD instructions

Similarly to the case of the `gemv`, if available on the processor SIMD (Single-Instruction Multiple-Data, that are instructions that perform the same operation in parallel on all elements of a small vector of data) can be used to improve the performance of the `symv` kernel in case the limiting factor is not the memory bandwidth from main memory.

Continuing the example, let us assume that the hypothetical processor has 2-wide vector units (i.e. each holding 2 FP numbers), and vector FMA and load instructions (operating on the 2-wide vectors) with the same latency and throughput of the scalar instructions used in Section 4.1.1.1.

As in the vector version of the 'T' `gemv` kernel, vectorization is used to compute several multiple loop iterations with a single instruction, instead of to compute several elements of the result vector with a single instruction as in the 'N' `gemv` kernel. The `symv` kernel computing a 4×1 sub-vector of y_t and a rank-4 update of y_n can be implemented using vector instructions as

$$\begin{array}{c|cccc} & \begin{bmatrix} a_{k0} \\ a_{h0} \end{bmatrix} & \begin{bmatrix} a_{k1} \\ a_{h1} \end{bmatrix} & \begin{bmatrix} a_{k2} \\ a_{h2} \end{bmatrix} & \begin{bmatrix} a_{k3} \\ a_{h3} \end{bmatrix} \\ \hline \begin{bmatrix} y_h^n \\ y_k^n \end{bmatrix} & \begin{bmatrix} y_h^n \Leftarrow a_{h0} \cdot x_0^n \\ y_k^n \Leftarrow a_{k0} \cdot x_0^n \end{bmatrix} & \begin{bmatrix} y_h^n \Leftarrow a_{h1} \cdot x_1^n \\ y_k^n \Leftarrow a_{k1} \cdot x_1^n \end{bmatrix} & \begin{bmatrix} y_h^n \Leftarrow a_{h2} \cdot x_2^n \\ y_k^n \Leftarrow a_{k2} \cdot x_2^n \end{bmatrix} & \begin{bmatrix} y_h^n \Leftarrow a_{h3} \cdot x_3^n \\ y_k^n \Leftarrow a_{k3} \cdot x_3^n \end{bmatrix} \\ \begin{bmatrix} x_h^t \\ x_k^t \end{bmatrix} & \begin{bmatrix} y_0^{t,0} \Leftarrow a_{h0} \cdot x_h^t \\ y_0^{t,1} \Leftarrow a_{k0} \cdot x_k^t \end{bmatrix} & \begin{bmatrix} y_1^{t,0} \Leftarrow a_{h1} \cdot x_h^t \\ y_1^{t,1} \Leftarrow a_{k1} \cdot x_k^t \end{bmatrix} & \begin{bmatrix} y_2^{t,0} \Leftarrow a_{h2} \cdot x_h^t \\ y_2^{t,1} \Leftarrow a_{k2} \cdot x_k^t \end{bmatrix} & \begin{bmatrix} y_3^{t,0} \Leftarrow a_{h3} \cdot x_h^t \\ y_3^{t,1} \Leftarrow a_{k3} \cdot x_k^t \end{bmatrix} \end{array} \tag{4.7}$$

(where $\Leftarrow$ has the usual meaning of short notation for the accumulator operator, and some of the square brackets have been removed to make the notation shorter: it is intended that all operations inside each pair of square brackets are vector operations on 2-wide vectors) that accounts for 8 FMAs, and is analogue to (4.4) implemented using scalar instructions. In fact, it is immediately evident that the 4 FMAs coming from the 'T' part are independent, but the 4 FMAs coming from the 'N' part are not. Even issuing the FMAs from the 'T' part between the dependent FMAs from the 'N' part, there are not enough independent instructions to keep the FMA pipeline busy, as on average it is possible to issue a FMA every 2 cycles, for a maximum performance of 50% of the vector full FP throughput.

In the vector case, it is possible to have enough independent FMA instructions by considering four consecutive iteration loops instead of two. In order to make the notation easier, the elements of the first, second, third and fourth iteration are denoted with $_h$, $_k$, $_l$ and $_p$. The resulting 16 FMAs computed in two consecutive

loops are

$$
\begin{array}{c|cccc}
 & \begin{bmatrix} a_{k0} \\ a_{h0} \end{bmatrix} & \begin{bmatrix} a_{k1} \\ a_{h1} \end{bmatrix} & \begin{bmatrix} a_{k2} \\ a_{h2} \end{bmatrix} & \begin{bmatrix} a_{k3} \\ a_{h3} \end{bmatrix} \\
\hline
\begin{bmatrix} y_h^n \\ y_k^n \end{bmatrix} & \begin{bmatrix} y_h^n \Leftarrow a_{h0} \cdot x_0^n \\ y_k^n \Leftarrow a_{k0} \cdot x_0^n \end{bmatrix} & \begin{bmatrix} y_h^n \Leftarrow a_{h1} \cdot x_1^n \\ y_k^n \Leftarrow a_{k1} \cdot x_1^n \end{bmatrix} & \begin{bmatrix} y_h^n \Leftarrow a_{h2} \cdot x_2^n \\ y_k^n \Leftarrow a_{k2} \cdot x_2^n \end{bmatrix} & \begin{bmatrix} y_h^n \Leftarrow a_{h3} \cdot x_3^n \\ y_k^n \Leftarrow a_{k3} \cdot x_3^n \end{bmatrix} \\
\begin{bmatrix} x_h^t \\ x_k^t \end{bmatrix} & \begin{bmatrix} y_0^{t,0} \Leftarrow a_{h0} \cdot x_h^t \\ y_0^{t,1} \Leftarrow a_{k0} \cdot x_k^t \end{bmatrix} & \begin{bmatrix} y_1^{t,0} \Leftarrow a_{h1} \cdot x_h^t \\ y_1^{t,1} \Leftarrow a_{k1} \cdot x_k^t \end{bmatrix} & \begin{bmatrix} y_2^{t,0} \Leftarrow a_{h2} \cdot x_h^t \\ y_2^{t,1} \Leftarrow a_{k2} \cdot x_k^t \end{bmatrix} & \begin{bmatrix} y_3^{t,0} \Leftarrow a_{h3} \cdot x_h^t \\ y_3^{t,1} \Leftarrow a_{k3} \cdot x_k^t \end{bmatrix} \\
\begin{bmatrix} y_l^n \\ y_p^n \end{bmatrix} & \begin{bmatrix} y_l^n \Leftarrow a_{l0} \cdot x_0^n \\ y_p^n \Leftarrow a_{p0} \cdot x_0^n \end{bmatrix} & \begin{bmatrix} y_l^n \Leftarrow a_{l1} \cdot x_1^n \\ y_p^n \Leftarrow a_{p1} \cdot x_1^n \end{bmatrix} & \begin{bmatrix} y_l^n \Leftarrow a_{l2} \cdot x_2^n \\ y_p^n \Leftarrow a_{p2} \cdot x_2^n \end{bmatrix} & \begin{bmatrix} y_l^n \Leftarrow a_{l3} \cdot x_3^n \\ y_p^n \Leftarrow a_{p3} \cdot x_3^n \end{bmatrix} \\
\begin{bmatrix} x_l^t \\ x_p^t \end{bmatrix} & \begin{bmatrix} y_0^{t,0} \Leftarrow a_{l0} \cdot x_l^t \\ y_0^{t,1} \Leftarrow a_{p0} \cdot x_p^t \end{bmatrix} & \begin{bmatrix} y_1^{t,0} \Leftarrow a_{l1} \cdot x_l^t \\ y_1^{t,1} \Leftarrow a_{p1} \cdot x_p^t \end{bmatrix} & \begin{bmatrix} y_2^{t,0} \Leftarrow a_{l2} \cdot x_l^t \\ y_2^{t,1} \Leftarrow a_{p2} \cdot x_p^t \end{bmatrix} & \begin{bmatrix} y_3^{t,0} \Leftarrow a_{l3} \cdot x_l^t \\ y_3^{t,1} \Leftarrow a_{p3} \cdot x_p^t \end{bmatrix}
\end{array}
\tag{4.8}
$$

It is possible to issue an independent vector FMA every clock cycle by issuing them using the same scheme (4.6) as in the scalar case for two following loop cycles. During the 16 clock cycles needed to issue all the 16 independent vector FMAs, 8 registers are loaded from memory. Therefore, if the matrix and vector elements are in L1 cache, it is theoretically possible to achieve the vector full FP throughput.

This scheme requires the same amount of (vector) FP registers as the scalar case requires scalar registers, i.e. 13-14 depending on the ISA. As a drawback, all instructions of 4 consecutive loop indexes need to be issued in a special order. In case of wider vectors, even more consecutive loops need to be considered at once, and this may affect the performance in case of small matrices.

The analysis about the use of contiguous memory and the panel-major matrix factor, and about the edge handling is analogue to the `gemv` case, found in Sections 4.1.2 and 4.1.3, and therefore it is not repeated.

4.3 Optimizing other level 2 BLAS routines

The computationally most expensive part of level 2 BLAS routines can be cast in terms of the `gemv` kernel, similarly as the computationally most expensive parts of level 3 BLAS and LAPACK routines can be cast in terms of the `gemm` kernel. In fact, the `gemv` kernel can be used to compute, upgrade or downgrade a sub-vector of the result vector with the product of one rectangular matrix and a vector. In the special case of level 2 BLAS routines with reuse factor of matrix elements equal to 2, it is convenient to use the `symv` kernel instead, even if these routines could be implemented using the `gemv` kernel.

The reminder of the section presents techniques to obtain high-performance level 2 BLAS routines based on the optimized `gemv` kernels, with special focus on small-scale performance.

4.3.1 Triangles and substitutions handling

The proposed level 2 BLAS routines for embedded optimization handle triangles and substitutions in the same way as the proposed level 3 BLAS and LAPACK routines handle the matrix counterpart of triangles and substitutions. Namely, specialized kernels are designed. In these kernels, the main loops is literally copied-and-pasted from the `gemv` and `symv` kernels, while specialized procedures before and after this loop take care of triangular matrices and substitutions. This approach requires the design of several specialized kernels, but once the `gemv` and `symv` kernels are available, they can be easily edited to get all other level 2 BLAS kernels.

4.3.2 Merging of linear algebra routines

Also in the case of level 2 BLAS routines, it is possible to employ merging of linear algebra routines in order to reduce overhead, and therefore improve performance.

Some examples of complex operations that are commonly found in embedded optimization and that can be easily merged are:

- The 'N' version of the `trsv` routine followed by the 'N' version of the `gemv` routine. It computes

$$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} \leftarrow \begin{bmatrix} A_0^{-1} x_0 \\ x_1 - A_1 \cdot A_0^{-1} x_0 \end{bmatrix} \quad \text{where} \quad A = \begin{bmatrix} A_0 \\ A_1 \end{bmatrix} \tag{4.9}$$

  and A_0 is a lower triangular invertible matrix and A_1 is a generic matrix. The matrix A could be the output of the `potrf` routine for $m > n$, as e.g. in the efficient implementation of the backward Riccati recursion in Chapter 8.

- The 'T' version of the `gemv` routine followed by the 'T' version of the `trsv` routine. It computes

$$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} \leftarrow \begin{bmatrix} A_0^{-T} \cdot (x_0 - A_1^T \cdot x_1) \\ x_1 \end{bmatrix} \quad \text{where} \quad A = \begin{bmatrix} A_0 \\ A_1 \end{bmatrix} \tag{4.10}$$

and A_0 is a lower triangular invertible matrix and A_1 is a generic matrix. The matrix A could be the output of the `potrf` routine for $m > n$, as e.g. in the efficient implementation of the backward Riccati recursion in Chapter 8.

4.3.3 Notable routines

This section reports the interface of notable routines as implemented for embedded optimization, together with the rationale behind the choice of the interfaces. All matrices are assumed to be in panel-major format.

4.3.3.1 trsv

There are two versions of the `trsv` routine, namely 'N' (where the matrix is considered as not-transposed) and 'T' (where the matrix is considered as transposed). The 'N' version of the routine is defined as

```
void dtrsv_n_lib(int m, int n, double *pA, int sda, \
    int use_inv_diag_A, double *inv_diag_A, double *x, double *y);
```

where $m \geq n$ and the top $n \times n$ sub-matrix A_0 of A is assumed to be invertible. If $m = n$, the routine is analogue to the `dtrsv` routine in standard BLAS. If $m > n$, the routine implements the merging of the `gemv` and `trsv` routines as defined in Section 4.3.2. The 'N' version computes the operation (4.9), while the version 'T' computes the operation (4.10). The matrix A is in the format as returned e.g. by the `potrf` routine proposed in Section 3.3.3. If used, the vector `inv_diag_A` has to contain the inverse of the diagonal elements of the matrix A_0: this vector is returned e.g. by the proposed `potrf` routine, avoiding the additional divisions.

4.3.3.2 gemv_nt

The `symv` kernel can be used to implement the 'N' and 'T' version of `gemv` in a single routine, when the matrix in the 'N' and 'T' versions of `gemv` is the same. This has the advantage of allowing for a reuse of matrix elements equal to 2, once they are moved into registers. The routine is defined as

```
void dgemv_nt_lib(int m, int n, double *pA, int sda, double *x_n, \
    double *x_t, int alg, double *y_n, double *y_t, double *z_n, \
    double *z_t);
```

The interface is analogue to the one of the `gemv` routines, with the difference that all vectors for both version have to be provided at once.

4.4 Performance of level 2 BLAS routines

This section contains performance plots for the level 2 BLAS routines that constitute the backbone of the optimization algorithms presented in Part II and Part III of this thesis. For each routine, three version are compared: the version implemented using the techniques presented in this thesis (HPMPC), the reference BLAS (Netlib BLAS 3.5.0), an optimized vendor BLAS (Intel's MKL 11.3) and an optimized open-source BLAS (OpenBLAS 0.2.15).

All routines are tested on the same two processors employed in Section 3.5, implementing Intel Ivy-Bridge and Intel Haswell micro-architectures respectively. The former supports the AVX ISA, while the latter supports the AVX2 and FMA ISAs, that are the best ISAs supported even in more recent micro-architectures such as Intel Broadwell and Intel Skylake. The two test machines have the same memory configuration, namely 8 GB of DDR3/DDR3L memory in dual-channel configuration (for a total data width of 128 bits), running at 1600 MHz, that gives a maximum bandwidth of 25.6 GB/s. Therefore, the difference in performance is solely due to the processors. These tests also show the current support state of the latest and previous latest x86_64 ISAs regarding FP.

The tests are performed in double precision, for squared matrices of size n between 4 and 300, in steps of 4, and are meant to evaluate the performance for matrix size of interest for embedded optimization applications. In all tests, only one thread is employed: therefore, the single-thread version of optimized BLAS libraries is considered.

4.4.1 Performance on Intel Ivy-Bridge micro-architecture

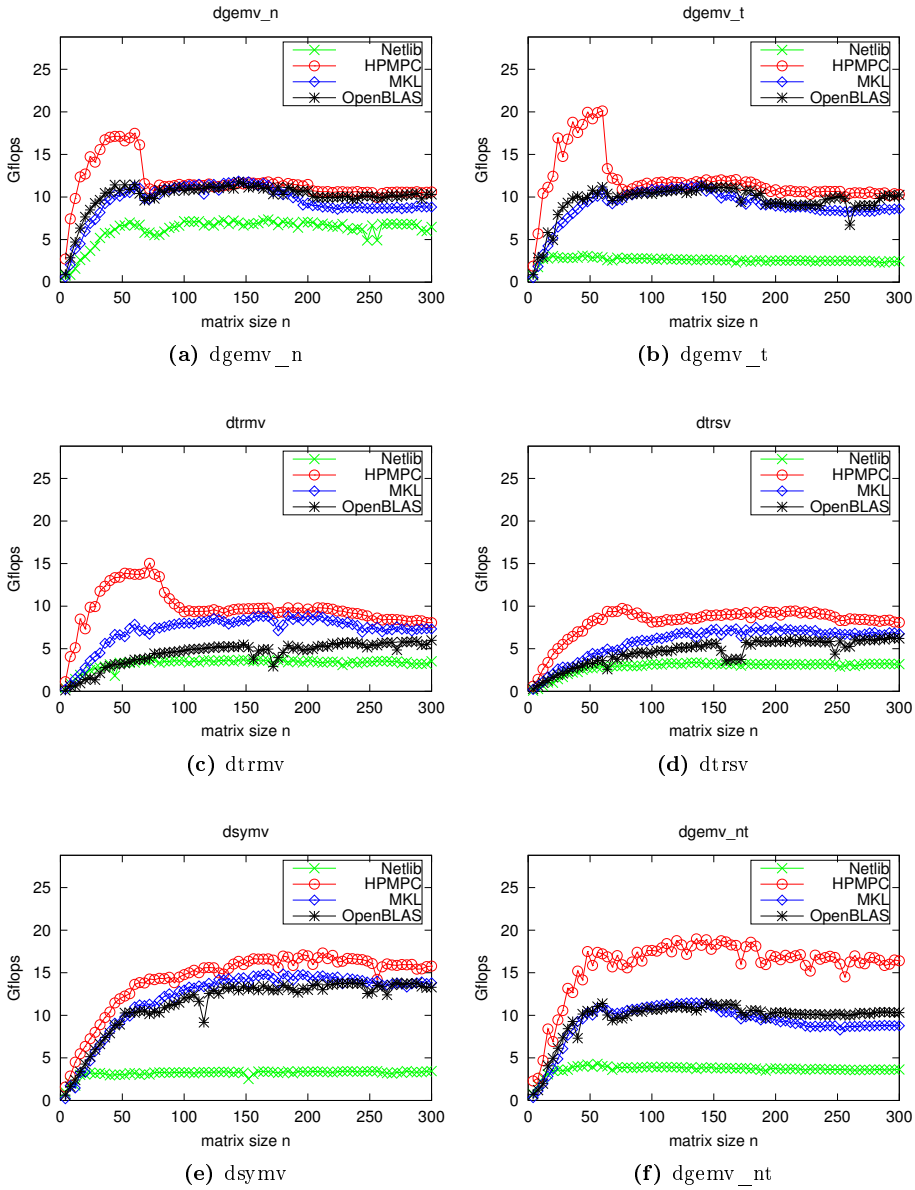
The performance plots are in Figure 4.1. The overall result is that in level 2 BLAS routines, if the matrix data is not already in cache, the cost to move the matrix data is the factor limiting performance. Most routines in HPMPC give very high performance if the matrix data is already in L1 cache (for squared

matrices of size up to size 64). The performance however decreases to the level of the other optimized BLAS version if the matrix data is in L2 cache (for squared matrices of size up to about 180), in L3 cache or main memory (the latter case not in figure). It is interesting to notice that there is a very little drop in performance when streaming data from L2 or L3 cache.

The `dgemv_nt` routine is a custom one and not part of BLAS. It performs the two general matrix-vector products, one with the matrix considered normal and one with the matrix considered transposed. Therefore, this operation can be performed by means of two calls to the `dgemv` routines in standard BLAS. The advantage of having a custom routine for this operation (that is employed in the computation of the residuals of the KKT system) is that every matrix element is reused twice once in registers. This gives a good performance advantage of the custom routine in HPMPC over the use of two calls to the `dgemv` routine in optimized BLAS. Reference BLAS performs rather poorly.

4.4.2 Performance on Intel Haswell micro-architecture

The performance plots are in Figure 4.2. Even more than in the case of level 3 BLAS and LAPACK routines, the recent AVX2 and FMA ISAs do not seem totally exploited in level 2 BLAS routines from optimized BLAS libraries. Therefore, generally the routines in HPMPC give some performance advantage over the corresponding routines in optimized BLAS libraries. In this micro-architecture, it is interesting to notice that, beside the performance drop when data has to be streamed from L2 cache, there is another performance drop when data has to the streamed from L3 cache. This performance drop was much smaller in the Ivy-Bridge architecture, hinting at the fact that, with respect to the Ivy-Bridge micro-architecture, the L1 and L2 cache bandwidth has doubled in the Haswell micro-architecture, while the L3 cache bandwidth looks the unchanged. Reference BLAS performs rather well on routines are not transposed, and extremely poorly on routines where the matrix is transposed (compare e.g. the performance of the 'N' and 'T' versions of the `dgemv` routine in Figures 4.2a and 4.2b). This is due to the fact that the 'N' version of the level 2 BLAS routines is based on the `scal` level 1 BLAS routine, that can be trivially vectorized, and that streams the result vector, and therefore having independent consecutive FMAs. Whereas the 'T' version of the level 2 BLAS routines is based on the `dot` level 1 BLAS routines, that is harder to vectorize since it requires final reduction, and that uses a single accumulation variable in consecutive iterations of the inner loops. Therefore, the 'N' versions can take advantage of the higher throughput given by the FMA ISA, while the 'T' versions suffer the higher latency (the FMA instruction has a latency of 5 cycles, while employing unfused instructions, the latency of the multiplication instruction can be hidden by the



(a) dgemv_n **(b)** dgemv_t

(c) dtrmv **(d)** dtrsv

(e) dsymv **(f)** dgemv_nt

Figure 4.1: Performance test for key level 2 BLAS routines on an Intel core i7 3520M processor (Ivy Bridge micro-architecture, supporting the AVX ISA).

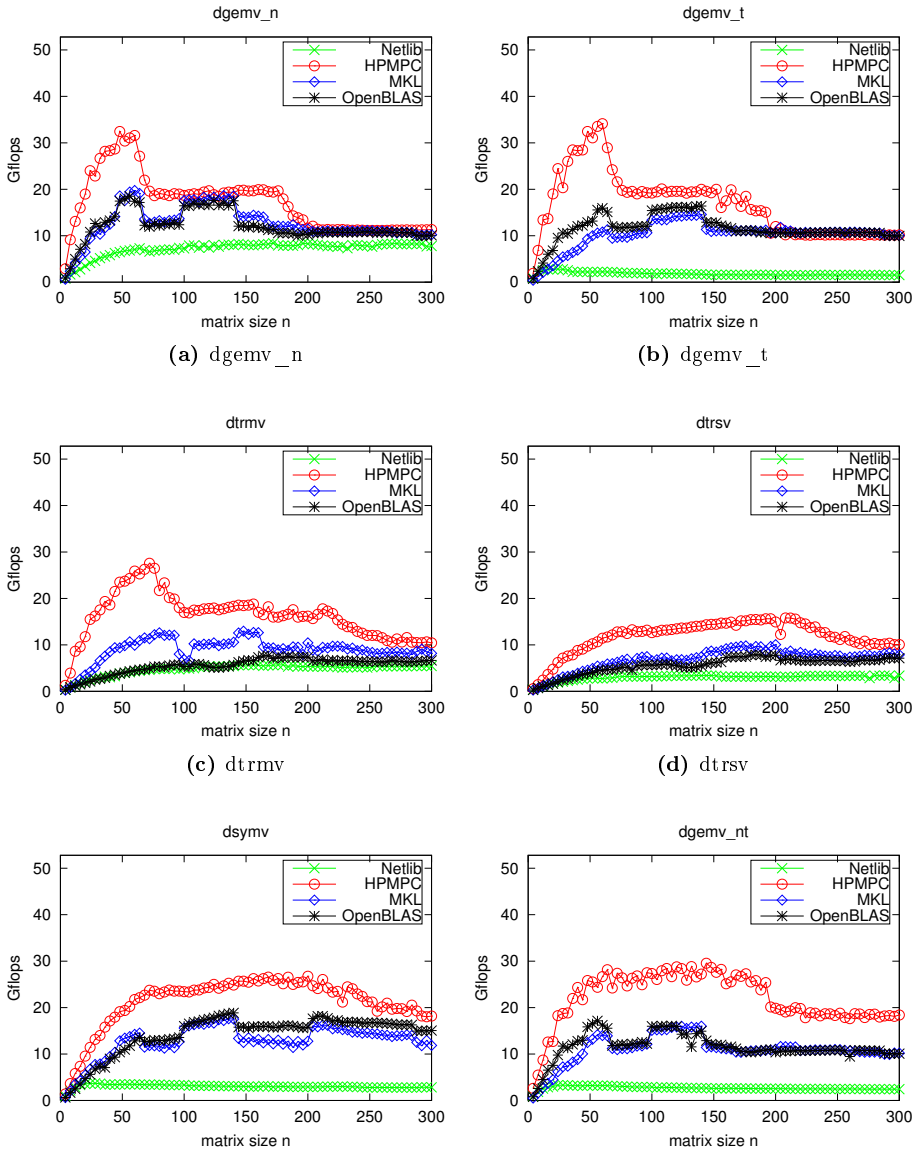
register renaming and the addition instruction has a latency of 3 cycles), and therefore give lower performance with the FMA ISA that with the AVX ISA.

4.5 Conclusion

This section extended to the level 2 BLAS linear algebra routines the implementation strategy proposed in Chapter 4. In case of level 2 BLAS routines, there is smaller room for optimization, since matrix elements have a reuse factor of $\mathcal{O}(1)$ (and typically exactly 1) and the streaming of data from main memory is often the bottleneck. However, if the matrices are already in cache (as it may be in the case of embedded optimization algorithms), the proposed implementation strategy gives a nice performance improvement.

In the case of level 2 BLAS routines, the bulk of the computation is cast in terms of the `gemv` kernel, with the notable exception of the `symv`-like routines, that can take advantage of a tailored kernel due to the reuse factor equal to 2.

Numerical tests confirm that, if the matrix data has to be streamed from L2 cache, L3 cache or main memory, there is little difference in performance between different implementation of level 2 BLAS routines, since the cost to move the matrix data is the factor limiting performance. However, if the matrix data is already present in L1 cache (or L2 for the Haswell micro-architecture), the routines in HPMPC give a nice performance advantage over the corresponding routines in optimized BLAS and LAPACK libraries. Reference BLAS routines (especially the 'T' versions) perform rather poorly also in case of level 2 BLAS routines.



(a) dgemv_n **(b)** dgemv_t

(c) dtrmv **(d)** dtrsv

(e) dsymv **(f)** dgemv_nt

Figure 4.2: Performance test for key level 2 BLAS routines on an Intel core i7 4800MQ processor (Haswell micro-architecture, supporting the AVX2 and FMA ISAs).

Chapter 5

Optimizing gemm kernels on different architectures

This chapter describes the practical implementation of the `gemm` kernel on a number of architectures. Code snippets are included, showing instruction and implementation methods characteristic of the different architectures.

Depending on the architecture, the code is implemented in either intrinsics or assembly. The choice between the two is a trade-off between easy of implementation and control over the resulting object code. Both of them generally give good control over the instructions that are present on the actual object code, and the choice between scalar and vector code.

Assembly code gives full control over instruction scheduling and register allocations. These features are particularly useful in case of in-order processor that do not implement register renaming. In fact, in this case the order of the instruction in the code corresponds exactly to the order in which they are executed in the processor. Therefore, if there are dependencies between two instructions, the processor stalls until the operands of the second instructions are available. Furthermore, register names have to be used carefully to avoid the introduction of dependencies.

Generally speaking, intrinsics (or intrinsic functions) are functions whose implementation is handled specifically by the compiler. In the considered framework,

the `icc`, `gcc` and `clang` compilers (among others) provide C intrinsics that are mapped directly into processor instructions (scalar and vector), and that operates on quantities that are mapped into processor registers. This gives an easy way to guide the compiler in the use of efficient instructions from C, without the need to write assembly code. As an example, this enables the explicit use of SIMD instructions, and of operations that are not directly available in the C language such as fused-multiply-add and shuffle. When intrinsics are employed, the compiler takes care of instruction scheduling and registers allocation. This generally works well with out-of-order processors that implement register renaming, but in case of in-order processor it may be preferable to have full control over these aspects. Therefore, in this chapter intrinsics are used only for architectures supporting out-of-order execution and register renaming.

The material contained in this chapter has diverse origins. Detailed description of many micro-architectures can be found in [1]. Useful instruction tables (containing e.g. latency, throughput and execution unit) for x86 and x86_64 architectures can be found in [2]. Examples of highly-optimized kernels for linear algebra can be found in the open-source OpenBLAS library [94] or in the open-sourc BLIS library [6].

Nonetheless, all kernels presented in this section have been originally tailored for the proposed implementation strategy for embedded optimization (e.g. they assume the panel-major matrix format). Some kernels (as e.g. `dgemm` and `sgemm` for the x86 Bonnell, or `sgemm` for the ARMv7A processors) significantly outperform the corresponding versions in OpenBLAS.

All performance plots in this chapter are scaled on the y-axis such that the top of the figure corresponds to the full FP performance. This makes it intuitive the evaluation of the performance: a good routine has a performance plot steadily close to the top of the figure.

5.1 x86

The x86 is a family of backward compatible instruction sets. Over the years, many extensions have been added.

In the following, by x86 it is meant a 32-bit CISC (Complex Instruction Set Computing) architecture. It features complex addressing modes and instructions where one of the source operands can be in memory. There are 8 GP (General Purpose) registers. On the FP side, the SSEx instruction sets up to SSE3 are considered in this thesis. These instruction sets operate on 8 128-bit wide

registers. The instructions take two-operands, and thus one of the two source operands has to be overwritten with the result.

Only natively 32-bit processors are considered. Therefore 64-bit processors operating in 32-bit mode are not considered. In 32-bit mode, the maximum amount of byte-addressable memory is limited to 4 GB, and the number of register names is limited to 8 GP and 8 FP. The memory limit does not affect performance in embedded optimization, but the smaller number of FP registers does. In particular, generally it is not possible to reach near full FP throughput if only 8 FP registers are employed, since latency of instructions can not be completely hidden.

5.1.1 Intel Bonnell (Atom)

The micro-architecture of the first generation of Atom processors is called Bonnell. Both 32-bit and 64-bit processors are based on this micro-architecture. Since the test machine considered in tests is 32-bit, only the 32-bit version is considered in this thesis.

The focus is on low cost and low power consumption processors. The Bonnell micro-architecture implements has an in-order dual-issue pipeline, and there is no register renaming nor speculative execution. As many modern x86 and x86-64 micro-architectures, the instructions are translated into simpler internal micro-operations. However, in the case of the Bonnell micro-architecture, the micro-operations are more CISC-like than RISC-like, as they can combine an ALU (Arithmetic Logic Unit) operation with a load or a store. Therefore the Bonnell micro-architecture has many similarities with the P5 micro-architecture of the original Pentium processor. There are 24 KB L1 data cache and 32 KB L1 instruction cache, plus an unified 512 KB L2 cache.

On the FP side, the SSE, SSE2 and SSE3 instruction sets are supported. The SSE instruction set contains mainly instructions for single-precision computation on 128-bit wide registers (each holding 4 single-precision FP numbers). The SSE2 instruction set contains mainly instructions for double-precision computation on 128-bit wide registers (each holding 2 double-precision FP numbers). The SSE3 instruction set contains instructions that allow to work horizontally in a register, e.g. adding or subtracting the elements in a vector register, or duplicating the lower element to the register. However, the SSE2 instruction set is implemented in a low power fashion, with some vector instruction split into two scalar ones.

The small number of registers, the fact that instructions have 2 operands, the

lack of a fused-multiply-add instruction and the lack of advanced features (such as out-of-order computation and register renaming) makes the optimization on this processor difficult: therefore, the use of inline assembly code is necessary to obtain good performance. Out of 8 available FP registers, 4 are employed as accumulation registers, and the other 4 are used to load A and B elements, and to hold intermediate results.

Both the dgemm and the sgemm kernels for x86 Bonnell are novel and an improvement over original Goto's kernels in OpenBLAS.

5.1.1.1 dgemm

The SSE2 instruction set provides support for 2-wide SIMD in double precision. However, the double-precision vector multiplication has a latency and a throughput of 9 cycles, making the vector version of `dgemm` effectively slower that the scalar version. Therefore, a scalar version of the code is considered in the following.

The double precision scalar multiplication has a throughput of 2 (i.e. an instruction can be issued every 2 clock cycles) and the scalar addition a throughput of 1. Since in the matrix-matrix multiplication there is an equal number of multiplications and additions, the multiplication is the limiting factor, and on average also the addition is issued every 2 clock cycles.

Out of the 8 FP registers, 4 can be used to hold a 2×2 sub-matrix of C, while the other 4 registers are used to hold elements from A and B, and intermediate results from the multiplications. A 2×2 kernel is employed, with 2 is the height of the panels in the panel-major matrix format.

The optimized code of an iteration over k is

```
1: addsd  %%xmm4, %%xmm2
2: movsd  16(%%edx), %%xmm4
3: mulsd  16(%%edx), %%xmm6

4: addsd  %%xmm7, %%xmm3
5: movsd  24(%%eax), %%xmm7
6: mulsd  24(%%eax), %%xmm4

7: addsd  %%xmm5, %%xmm1
8: movsd  24(%%edx), %%xmm5
```

```
 9: mulsd  24(%%edx), %%xmm7

10: addsd  %%xmm6, %%xmm0
11: movsd  32(%%eax), %%xmm6
12: mulsd  16(%%eax), %%xmm5
```

where the right operand is overwritten with the result of the instruction. Registers `xmm0` to `xmm3` are used to hold a submatrix of D,

$$\begin{bmatrix} \texttt{xmm0} & \texttt{xmm1} \\ \texttt{xmm2} & \texttt{xmm3} \end{bmatrix} \leftarrow \begin{bmatrix} d_{00} & d_{01} \\ d_{10} & d_{11} \end{bmatrix}$$

The instructions `addsd` and `mulsd` perform respectively the scalar double-precision FP addition and multiplication. The instruction `movsd` can load, store or move between registers a scalar double-precision FP number. Since there are not fused-multiply-accumulate nor out-of-order execution, the general idea behind the optimization is to hide multiplications latency by having enough independent instructions between each multiplication and the relative addition, with the constraint of the limited number of registers. In this case, the result of the multiplication at line 3 is held in the register `xmm6`, and the relative addition takes place at line 10: in this way a latency of 5 cycles can be completely hidden.

At line 3, the choice of taking one of the operands from memory instead of from the register `xmm4` loaded at the previous line is justified by the fact that in this way there are less dependent instructions, and that the maximum capacity of one load per cycle is not exceeded.

A performance test shows that the maximum performance is attained when the data fits the L1 cache size, hinting at the fact that hardware data prefetch is not implemented. If software prefetch is employed, the high performance is attained also for data fitting into L2 cache, since the memory bandwidth between L2 and L1 cache is big enough to feed the processor, and the latency is hidden by moving the data into L1 cache before it is needed.

5.1.1.2 sgemm

The SSE instruction set provides support for 4-wide SIMD in single precision. Both 4-wide vector multiplication and addition have a throughput of 2, while scalar versions have a throughput of 1. Therefore in single precision it is advantageous to use the vector version, that has twice the full FP throughput.

Again, out of the 8 FP registers, 4 can be used to hold a sub-matrix of D and 4 for intermediate results. Since each register can hold 4 floats, a 4×4 sub-matrix

of D can be hold in 4 registers. This means that a 4×4 kernel is employed, and the panel height is 4.

The optimized code of an iteration over k is

```
 1: addps   %%xmm1, %%xmm5
 2: movaps  %%xmm0, %%xmm1
 3: shufps  $0, %%xmm0, %%xmm0
 4: mulps   16(%%eax), %%xmm0

 5: addps   %%xmm2, %%xmm6
 6: movaps  %%xmm1, %%xmm2
 7: shufps  $85, %%xmm1, %%xmm1
 8: mulps   16(%%eax), %%xmm1

 9: addps   %%xmm3, %%xmm7
10: movaps  %%xmm2, %%xmm3
11: shufps  $170, %%xmm2, %%xmm2
12: mulps   16(%%eax), %%xmm2

13: addps   %%xmm0, %%xmm4
14: movaps  32(%%edx), %%xmm0
15: shufps  $255, %%xmm3, %%xmm3
16: mulps   16(%%eax), %%xmm3
```

Registers `xmm4, xmm5, xmm6, xmm7` hold respectively the first, second, third and fourth column of the 4×4 sub-matrix of D, each consisting of 4 elements:

$$\texttt{xmm4} \leftarrow \begin{bmatrix} d_{00} \\ d_{10} \\ d_{20} \\ d_{30} \end{bmatrix}, \quad \texttt{xmm5} \leftarrow \begin{bmatrix} d_{01} \\ d_{11} \\ d_{21} \\ d_{31} \end{bmatrix}, \quad \texttt{xmm6} \leftarrow \begin{bmatrix} d_{02} \\ d_{12} \\ d_{22} \\ d_{32} \end{bmatrix}, \quad \texttt{xmm7} \leftarrow \begin{bmatrix} d_{03} \\ d_{13} \\ d_{23} \\ d_{33} \end{bmatrix}.$$

Instructions `addps` and `mulps` perform respectively the vector (packed) single-precision FP addition and multiplication. Instruction `movps` can load, store or move between registers a vector of 4 packed single-precision FP numbers. Instruction `shufps` shuffles the content of vectors of 4 packed single-precision FP numbers.

In the x86 (and in the following x86_64) architecture, a scalar instruction can operate only on the lower element of a register, while a vector instruction can operate only on all element of a register at the same time (and not on sub-vectors). Thus shuffle instructions are needed to reorder the elements within a

register. In particular, in the above code, assuming that before line 1 the `xmm0` register has been loaded with the vector

$$\texttt{xmm0} \leftarrow \begin{bmatrix} b_{0k} \\ b_{1k} \\ b_{2k} \\ b_{3k} \end{bmatrix}$$

then the shuffle instruction at line 3 broadcasts the element b_{0k} to all elements of the vector `xmm0`, and similarly shuffle instructions at lines 7, 11 and 15 broadcast respectively the elements b_{1k}, b_{2k} and b_{3k} to all elements of vectors `xmm1`, `xmm2` and `xmm3`. The move instructions at lines 2, 6 and 10 save the original content of register `xmm0` (before the shuffle instructions destroy its value) into registers `xmm1`, `xmm2` and `xmm3` as soon as they are free after the immediately previous addition. The move instruction at line 14 loads `xmm0` with a new vector from matrix B.

The element from the A matrix in each multiplication instruction has to be fetch from memory even if it is the same for all 4 multiplications, since the small number of registers (8) prevents the use of an extra register to hold its value.

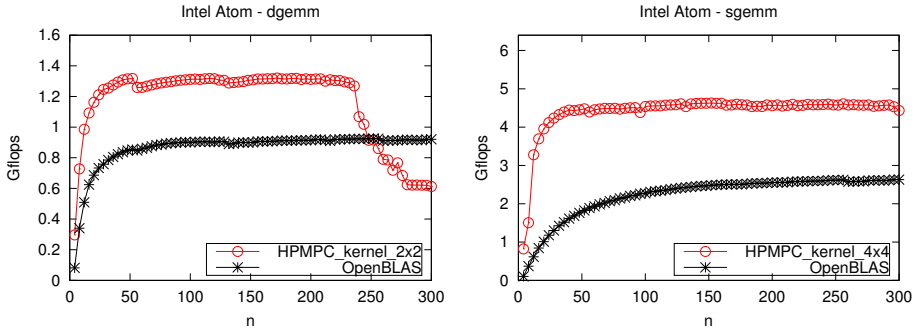
Also in single precision software prefetch has to be employed to obtain the best performance for data fitting L2 cache.

5.1.1.3 Results

The test processor is the Intel Atom N270, with one core at 1.6 GHz. As shown in Figure 5.1, the `dgemm` and `sgemm` kernels attain a best performance of respectively 1.34 Gflops (83% of full FP throughput) and 4.63 Gflops (72% of full FP throughput). There is a big performance boost over the OpenBLAS implementation, that has a best performance respectively of 0.92 Gflops (57%) and 2.63 Gflops (41%). This is due to the better instruction scheduling of the proposed implementation schemes, compared to the kernels in OpenBLAS.

5.2 x86_64

The x86_64 is the 64-bit version of the x86 instruction set. The original specification has been created by AMD and released in 2000.



(a) Atom dgemm. **(b)** Atom sgemm.

Figure 5.1: Performance test of different implementations of `gemm` for squared matrices of size $n \times n$, $n \in [4, 300]$, on an Intel Atom N270, code compiled with gcc 4.6.3. Peak performance in double (single) precision is $1 \cdot 1.6 = 1.6$ Gflops ($4 \cdot 1.6 = 6.4$ Gflops).

In the following, by x86-64 it is meant a 64-bit CISC architecture. The amount of byte-addressable memory is now equal to 2^{64}, that is well beyond the amount of memory available on today computers. There are 16 GP registers. On the FP side, there are several instruction sets such as the SSEx ISAs up to SSE4.2, the AVX and AVX2 ISA (all supporting 16 FP registers), and the forthcoming AVX512 ISA (supporting 32 FP registers). Generally speaking, the larger number of registers makes optimization easier. In particular, in most architectures, near full FP throughput can be achieved only in 64-bit mode.

5.2.1 Intel Core

The Core architecture (codename Merom) arrived to market in 2006 as a "tock" (a new micro-architecture). Is is implemented on Intel 65 nm process. The following "tick" (codename Penryn) is a die shrink implemented on Intel 45 nm process.

The Core micro-architecture is an evolution of the older P6 micro-architecture (used in the Pentium Pro, Pentium II, Pentium III), after the false step represented by the NetBurst micro-architecture (used in the Pentium IV). While the NetBurst micro-architecture focused on improving performance by achieving high frequencies (e.g. by employing very long pipelines) at the expenses of efficiency and power consumption, the Core micro-architectures focuses on

efficiency and increasing the work done per clock cycle.

On the ISA side, there are not many new features compared to NetBurst. Merom supports SSEx ISAs up to SSSE3. Penryn supports also SSE4.1, that contains specialized instructions and therefore has limited impact on FP performance: in this framework the only notable new instruction is `blend`, that however in Penryn has the same latency and throughput of the more general `shuffle` instruction part of the SSE and SSE2 ISAs. All SSEx ISAs have two-operand instructions.

However, the Core micro-architecture considerably differs from the NetBurst micro-architecture. The Core micro-architecture is a 64-bit superscalar processor that can issue 4 instructions per clock cycle. It is a dual-core design, that drops the SMT (simultaneous multi-thread) introduced in the NetBurst micro-architecture, and re-introduced in the following Nehalem micro-architecture. It supports speculative and out-of-order execution, and register renaming on both GP and FP registers. There are 32 KB of 8-way associative L1 instruction cache and 32 KB of 8-way associative L1 data cache per core. There is also a L2 cache shared between CPU cores.

On the FP side, the Core micro-architecture has 128-bit wide FP execution units, and therefore it can issue 128-bit wide SIMD instructions in one clock cycle (that are therefore fully pipelined). As a comparison, the NetBurst micro-architecture has 64-bit wide FP execution units, and therefore 128-bit wide SIMD instructions are internally split into two instructions. The Core micro-architecture can sustain a 128-bit FP multiplication and a 128-bit FP addition every clock cycle. However, FP shuffles are issued on the same execution ports as the multiplication (`shufpd` and addition `shufps`, and this seriously affects performance of the `gemm` kernel. Therefore, the universal shift instruction `pshufd` is preferred, since it is issued on a different execution port, and can therefore co-issued with a FP multiplication and a FP addition at each clock cycle. The Core micro-architecture has 1 128-bit load unit and 1 128-bit store unit.

In multiplication has a latency of 5 clock cycles in double precision and 4 in single precision, and a throughput of 1 in both precisions, while addition has a latency of 3 clock cycles and a throughput of 1 in both precisions. Since the processor can perform register renaming, at least 3 registers have to be used as accumulation registers, to hide the addition latency. However, in practice it is convenient to use as more registers in order to increase the reuse factor and therefore reduce the number of memory operations. Since there are 16 FP registers in both single and double precision, in the `gemm` implementation scheme 8 registers are used to hold a sub-matrix of C, while the other 8 registers are used to hold elements from A and B, and intermediate results.

5.2.1.1 dgemm

Each 128-bit register can hold 2 double-precision FP numbers. In double precision, both the multiplication and the addition instructions have a throughput of 1 instruction per clock cycle, and at each clock cycle both a multiplication and an addition can be issued. Therefore in double precision the full FP throughput of the Core architecture is 4 flops per clock cycle.

In the `dgemm` optimization, 8 out of 16 registers can be used to hold a 4×4 sub-matrix of D. Thus a 4×4 kernel is employed, with panel height of $b_s = 4$.

The Core architecture is out of order, that means that instructions can be reordered on the fly by the processor. However, the Reorder Buffer (ROB) size is rather limited, and therefore instructions order can not differ too much compared to the optimal one. This is an argument in favor to the use of assembly instead of C intrinsics. Furthermore, for performance reasons it is convenient to use the universal shuffle instruction `pshufd` instead of the FP version `shufpd`. The intrinsic function making use of the `pshufd` instruction is `_mm_shuffle_epi32`, that operates on 32-bit integers (therefore cast would be needed) and has a single register operand, while the actual `pshufd` instruction has two register operands.

Therefore, the code is better written in assembly. The optimized code of an iteration over k is:

```
 1: addpd   %%xmm6, %%xmm10
 2: movaps  16(%%rbx), %%xmm6
 3: addpd   %%xmm3, %%xmm14
 4: movaps  %%xmm2, %%xmm3
 5: pshufd  $0x4e, %%xmm2, %%xmm7
 6: mulpd   %%xmm0, %%xmm2
 7: mulpd   %%xmm1, %%xmm3

 8: addpd   %%xmm4, %%xmm11
 9: addpd   %%xmm5, %%xmm15
10: movaps  %%xmm7, %%xmm5
11: mulpd   %%xmm0, %%xmm7
12: mulpd   %%xmm1, %%xmm5

13: addpd   %%xmm2, %%xmm8
14: movaps  32(%%rbx), %%xmm2
15: addpd   %%xmm3, %%xmm12
16: movaps  %%xmm6, %%xmm3
```

```
17: pshufd  $0x4e, %%xmm6, %%xmm4
18: mulpd   %%xmm0, %%xmm6
19: mulpd   %%xmm1, %%xmm3

20: addpd   %%xmm7, %%xmm9
21: addpd   %%xmm5, %%xmm13
22: movaps  %%xmm4, %%xmm5
23: mulpd   %%xmm0, %%xmm4
24: movaps  32(%%rax), %%xmm0
25: mulpd   %%xmm1, %%xmm5
26: movaps  48(%%rax), %%xmm1
```

The registers `%%xmm8` to `%%xmm15` are used as accumulation registers, holding a permutation of a 4×4 sub-matrix of D. Namely, the registers `%%xmm8` to `%%xmm11` hold the top 2×4 sub-matrix, as

$$\texttt{xmm8} \leftarrow \begin{bmatrix} d_{00} \\ d_{11} \end{bmatrix}, \quad \texttt{xmm9} \leftarrow \begin{bmatrix} d_{01} \\ d_{10} \end{bmatrix}, \quad \texttt{xmm10} \leftarrow \begin{bmatrix} d_{02} \\ d_{13} \end{bmatrix}, \quad \texttt{xmm11} \leftarrow \begin{bmatrix} d_{03} \\ d_{12} \end{bmatrix}$$

and similarly for registers `%%xmm12` to `%%xmm15`, holding the bottom 2×4 sub-matrix.

The other registers are used to hold elements from A and B, and to hold intermediate results.

5.2.1.2 sgemm

Each 128-bit register can hold 4 single-precision FP numbers. In single precision, both the multiplication and the addition instructions have a throughput of 1 instruction per clock cycle, and at each clock cycle both a multiplication and an addition can be issued. Therefore in single precision the full FP throughput of the Core architecture is 8 flops per clock cycle.

In the `sgemm` optimization, 8 out of 16 registers can be used to hold a 8×4 sub-matrix of D. Thus a 8×4 kernel is employed, with panel height of $b_s = 4$.

The arguments in favor to the choice of assembly over C intrinsics in the double precision case still hold in the single precision case. The optimized code of an iteration over k is:

```
1: addps   %%xmm6, %%xmm10
```

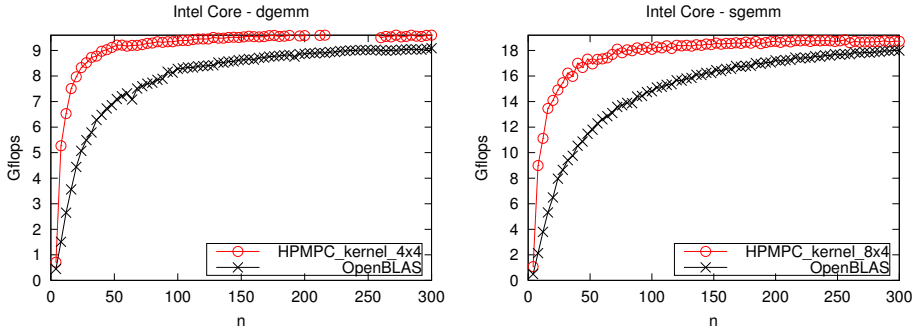
```
 2: addps   %%xmm3, %%xmm14
 3: movaps  %%xmm2, %%xmm3
 4: pshufd  $0x39, %%xmm2, %%xmm7
 5: mulps   %%xmm0, %%xmm2
 6: mulps   %%xmm1, %%xmm3

 7: addps   %%xmm4, %%xmm11
 8: addps   %%xmm5, %%xmm15
 9: movaps  %%xmm7, %%xmm5
10: pshufd  $0x39, %%xmm7, %%xmm6
11: mulps   %%xmm0, %%xmm7
12: mulps   %%xmm1, %%xmm5

13: addps   %%xmm2, %%xmm8
14: movaps  16(%%rbx), %%xmm2
15: addps   %%xmm3, %%xmm12
16: movaps  %%xmm6, %%xmm3
17: pshufd  $0x39, %%xmm6, %%xmm4
18: mulps   %%xmm0, %%xmm6
19: mulps   %%xmm1, %%xmm3

20: addps   %%xmm7, %%xmm9
21: addps   %%xmm5, %%xmm13
22: movaps  %%xmm4, %%xmm5
23: mulps   %%xmm0, %%xmm4
24: movaps  16(%%rax), %%xmm0
25: mulps   %%xmm1, %%xmm5
26: movaps  16(%%rcx), %%xmm1
```

The implementation scheme is analogue to the double precision one, just ported to single precision. The registers `%%xmm8` to `%%xmm15` are used as accumulation registers, holding a permutation of a 8×4 sub-matrix of D. Namely, the registers `%%xmm8` to `%%xmm11` hold the top 4×4 sub-matrix, as

$$\texttt{xmm8} \leftarrow \begin{bmatrix} d_{00} \\ d_{11} \\ d_{22} \\ d_{33} \end{bmatrix}, \quad \texttt{xmm9} \leftarrow \begin{bmatrix} d_{01} \\ d_{12} \\ d_{23} \\ d_{30} \end{bmatrix}, \quad \texttt{xmm10} \leftarrow \begin{bmatrix} d_{02} \\ d_{13} \\ d_{20} \\ d_{31} \end{bmatrix}, \quad \texttt{xmm11} \leftarrow \begin{bmatrix} d_{03} \\ d_{10} \\ d_{21} \\ d_{32} \end{bmatrix}$$

and similarly for registers `%%xmm12` to `%%xmm15`, holding the bottom 4×4 sub-matrix.

The other registers are used to hold elements from A and B, and to hold intermediate results.



(a) Intel Core dgemm.

(b) Intel Core sgemm.

Figure 5.2: Performance test of `gemm` kernel for squared matrices of size $n \times n$, $n \in [4, 300]$, on an Intel Core, code compiled with gcc 4.8.4. Peak performance in double (single) precision is $4 \cdot 2.4 = 9.6$ Gflops ($8 \cdot 2.4 = 19.2$ Gflops).

5.2.1.3 Results

The test machine is a laptop equipped with the Intel Core 2 Duo P8600 processor running at a nominal maximum frequency of 2.4 GHz, and equipped with 3 MB L2 cache. This processor is an implementation of the Penryn micro-architecture. Figure 5.2 contains the performance plot of the `gemm` routine. The `dgemm` kernel attains a maximum performance of 9.68 Gflops, that is 100.8% of full FP throughput: therefore it looks like the processor is running slightly faster than the nominal frequency. The `sgemm` kernel attains a maximum performance of 18.8 Gflops, that is 98% of the full FP throughput. As a reference, OpenBLAS attains a maximum performance of 9.09 Gflops (94.6%) in double precision and 18.03 Gflops (93.9%) in single precision. More importantly than the best absolute performance, the proposed implementation attains a very high performance already for small matrices, being around 50% for matrices of size 8 and very close to the full throughput for matrices of size 16.

5.2.2 Intel Nehalem

The Nehalem architecture arrived to market in 2008 as a "tock" (a new micro-architecture). It is implemented on the same Intel 45 nm planar process of the Penryn architecture, that is a "tick" (shrink to a new process technology) of the Merom micro-architecture. The following Westmere architecture is a "tick" of

Nehalem, making use of Intel 32 nm planar process.

On the ISA side, Nehalem does not introduce many improvements. It introduces the SSE4.2 ISA, that is of no interest in linear algebra routines implementation.

The Nehalem micro-architecture shares with its predecessor (Core) the fact that it is a 64-bit superscalar processor that can issue 4 instructions per clock cycle. It re-introduces SMT with two threads per physical core. It supports speculative and out-of-order execution, and register renaming on both GP and FP registers. There are 32 KB of 4-way associative L1 instruction cache and 32 KB of 8-way associative L1 data cache per core, plus 256 KB of 8-way associative L2 cache that is now per core. It also introduces a L3 cache shared between CPU cores.

The Nehalem introduces big improvements over Core micro-architecture on all aspects except the execution units. Therefore, the full FP throughput is the same as the Core micro-architecture. Furthermore, since the proposed linear algebra implementation for embedded optimization does not explicitly considers caches, TLBs or memory features, the implementation scheme is the same developed for the Core micro-architecture. There are only two small differences: a `kernel` specially designed for the Nehalem micro-architecture can be slightly simplified by using the FP shuffle instructions `shufpd` and `shufps` (now issued on a different port compared to FP multiplication and FP addition) instead of the universal shuffle instruction `pshufd`. This, together with the bigger reorder windows, should allow an intrinsics version of the code to obtain reasonable good performance, without the need for inline assembly. Another difference is that the `blend` instruction can be issued on two ports, and therefore long sequences of `blend` instructions can be processed faster. However, the differences are small, and the implementation scheme for the Core micro-architecture gives excellent performance.

5.2.3 Intel Sandy-Bridge

The Sandy-Bridge architecture arrived to market in 2011 as a "tock" (a new micro-architecture). It is implemented on the same Intel 32 nm planar process of the Westmere architecture, that is a "tick" (shrink to a new process technology) of the Nehalem micro-architecture. Sandy-Bridge is found in 2nd generation Intel Core processors. The following Ivy-Bridge architecture (found in 3rd generation Intel Core processors) is a "tick" of Sandy-Bridge, making use of Intel 22 nm FinFET process.

On the ISA side, Sandy-Bridge presents many improvements. The change most useful to numerical code is the introduction of the AVX ISA (providing 256-bit

wide FP SIMD). This ISA doubles the width of the various SSE (i.e. SSE, SSE2, SSE3, SSE4.1, SSE4.2) ISAs, and introduces 3 and 4 operands instructions (while the various SSE ISAs have 2 operands instructions).

The Sandy-Bridge micro-architecture shares with its predecessor (Nehalem) the fact that it is a 64-bit superscalar processor that can issue 4 instructions per clock cycle. It supports two threads per physical core, speculative and out-of-order execution, and register renaming on both GP and FP registers. There are 32 KB of 8-way associative L1 instruction cache and 32 KB of 8-way associative L1 data cache per core, plus 256 KB of 8-way associative L2 cache per core. There is also a L3 cache shared between CPU cores.

On the FP side, the Sandy-Bridge architectures introduces the new AVX instruction set, that extends the FP registers from 128-bit to 256-bit, and introduces new 3 and 4 operands instructions. There are 6 execution ports. In particular, there are separate ports for SIMD multiplication, SIMD addition, and SIMD shuffle, allowing a 256-bit wide multiplication, a 256-bit wide addition and a 256-bit wide shuffle to be co-issued at each clock cycle: this doubles the full FP throughput compared to the previous micro-architecture. In order to feed the wider 256-bit AVX units, the Sandy-Bridge micro-architecture has 2 128-bit load units and 1 128-bit store unit. However, it has only two address generation units, and therefore it can sustain two 128-bit loads or one 128-bit load and a 128-bit store per clock cycle. This means that the Sandy-Bridge micro-architecture can sustain the load of a 256-bit wide register per clock cycle, but it can not sustain the store of a 256-bit wide register per clock cycle. As a comparison, the Nehalem micro-architecture has 1 128-bit load unit and 1 128-bit store unit, and it can sustain a 128-bit load and a 128-bit store per clock cycle.

In both single and double precision, multiplication has a latency of 5 clock cycles and a throughput of 1, while addition has a latency of 3 clock cycles and a throughput of 1. Since the processor can perform register renaming, at least 3 registers have to be used as accumulation registers, to hide the addition latency. However, in practice it is convenient to use as more registers in order to increase the reuse factor and therefore reduce the number of memory operations. Since there are 16 FP registers in both single and double precision, in the `gemm` implementation scheme 8 registers are used to hold a sub-matrix of C, while the other 8 registers are used to hold elements from A and B, and intermediate results.

5.2.3.1 dgemm

Each 256-bit register can hold 4 double-precision FP numbers. In double precision, both the multiplication and the addition instructions have a throughput of 1 instruction per clock cycle, and at each clock cycle both a multiplication and an addition can be issued. Therefore in double precision the full FP throughput of the Sandy-Bridge architecture is 8 flops per clock cycle.

In the `dgemm` optimization, 8 out of 16 registers can be used to hold a 8×4 sub-matrix of D. Thus a 8×4 kernel is employed, with panel height of $b_s = 4$.

The Sandy-Bridge architecture is out of order, that means that instructions can be reordered on the fly by the processor. Therefore the order of instruction is not critical, and the compiler can take care of instruction scheduling. Furthermore, the Sandy-Bridge architecture implements register renaming. This means that a single register name can be used for all intermediate results, since it is mapped on different physical registers.

Therefore, the code can be safely written using intrinsics. The optimized code of an iteration over k is:

```
 1: tmp = _mm256_mul_pd( a_0123, b_0 );
 2: b_1 = _mm256_shuffle_pd( b_0, b_0, 0x5 ); // b_1032
 3: A_0 = _mm256_load_pd( &A0[4] ); // prefetch
 4: d_0 = _mm256_add_pd( d_0, tmp );
 5: tmp = _mm256_mul_pd( a_4567, b_0 );
 6: b_0 = _mm256_load_pd( &B[4] ); // prefetch
 7: d_4 = _mm256_add_pd( d_4, tmp );

 8: tmp = _mm256_mul_pd( a_0123, b_1 );
 9: b_2 = _mm256_permute2f128_pd( b_1, b_1, 0x1 ); // b_3210
10: A_4 = _mm256_load_pd( &A1[4] ); // prefetch
11: d_1 = _mm256_add_pd( d_1, tmp );
12: tmp = _mm256_mul_pd( a_4567, b_1 );
13: d_5 = _mm256_add_pd( d_5, tmp );

14: tmp = _mm256_mul_pd( a_0123, b_2 );
15: b_1 = _mm256_shuffle_pd( b_2, b_2, 0x5 ); // b_2301
16: d_3 = _mm256_add_pd( d_3, tmp );
17: tmp = _mm256_mul_pd( a_4567, b_2 );
18: d_7 = _mm256_add_pd( d_7, tmp );

19: tmp = _mm256_mul_pd( a_0123, b_1 );
```

```
20: d_2 = _mm256_add_pd( d_2, tmp );
21: tmp = _mm256_mul_pd( a_4567, b_1 );
22: d_6 = _mm256_add_pd( d_6, tmp );
```

The registers `d_0` to `d_7` are the accumulation registers, holding a permutation of a 8×4 sub-matrix of D. Namely, the registers `d_0` to `d_3` contain the permutation of the upper 4×4 block, as

$$\texttt{d\_0} \leftarrow \begin{bmatrix} d_{00} \\ d_{11} \\ d_{22} \\ d_{33} \end{bmatrix}, \quad \texttt{d\_1} \leftarrow \begin{bmatrix} d_{01} \\ d_{10} \\ d_{23} \\ d_{32} \end{bmatrix}, \quad \texttt{d\_2} \leftarrow \begin{bmatrix} d_{03} \\ d_{12} \\ d_{21} \\ d_{30} \end{bmatrix}, \quad \texttt{d\_3} \leftarrow \begin{bmatrix} d_{02} \\ d_{13} \\ d_{20} \\ d_{31} \end{bmatrix}$$

and similarly for the registers `d_4` to `d_7`, holding the lower 4×4 block. The use of this permutation avoids the saturation of the load unit. In fact, it allows to reduce the number of load instructions compared to the scheme that employs the `broadcast` instruction to fill the `b_0` vector with 4 copies of the b_{0k} element (similarly for the other elements). If the scheme employing `broadcast` instructions is considered, one `broadcast` instruction is needed at each clock cycle to load B elements, plus one `load` instruction every 4 clock cycles to load A elements. This exceeds the micro-architecture load issue capability of 1 256-bit load per clock cycle.

On the contrary, in the employed scheme, the extra load instructions are replaced by `shuffle` and `permute2f128` instructions, that can be issued in parallel with multiplication, addition and load instructions. At the end, the correct permutation can be recovered by means of two layers of `blend` instructions before storing the result.

At the beginning of the k iteration, the registers `a_0`, `a_4` and `b_0` contain the elements

$$\begin{matrix} \texttt{a\_0} \leftarrow \\ \\ \texttt{a\_1} \leftarrow \end{matrix} \begin{bmatrix} a_{0k} \\ a_{1k} \\ a_{2k} \\ a_{3k} \end{bmatrix} \begin{bmatrix} a_{4k} \\ a_{5k} \\ a_{6k} \\ a_{7k} \end{bmatrix}, \qquad \texttt{b\_0} \leftarrow \begin{bmatrix} b_{0k} \\ b_{1k} \\ b_{2k} \\ b_{3k} \end{bmatrix}$$

prefetched during the previous iteration. During the k-th iteration, the registers `A_0`, `A_1` and `b_0` are prefetched with the A and B elements used in the following iteration $k + 1$.

5.2.3.2 sgemm

Each 256-bit register can hold 8 single-precision FP numbers. In single precision, both the multiplication and the addition have a throughput of 1 instruction per clock cycle, and at each clock cycle both a multiplication and an addition can be issued. Therefore, in single precision the full FP throughput is 16 flops per clock cycle.

The panel height is chosen as $b_s = 8$, that is equal to the FP registers size in floats. In the `sgemm` optimization, 8 out of 16 registers can be used to hold a sub-matrix of D.

A natural choice would be to hold a 8×8 sub-matrix of D, such that each element from A and B is reused 8 times once in registers. However, in practice this implementation scheme is found to be sub-optimal, and the reason seems to lie in the saturation of the shuffle unit, since this scheme requires to shuffle the B vector at each clock cycle.

An alternative scheme holds a 16×4 sub-matrix of D. This allows to reuse the shuffled element of B in two consecutive clock cycles, reducing pressure on the shuffle unit. As a drawback, this scheme employs only the upper 4 elements the rows in each panel (that has an height of 8 elements). This is sub-optimal since a cache line is not fully utilized once moved into L1 cache. The optimal solution (i.e. employ different panel heights for the A and B matrix) can not be employed in the proposed linear algebra implementation scheme for embedded optimization, since all matrices need to have the same panel height value. Anyhow, in practice this does not seem to affect performance.

As discussed in the `dgemm` case, the code can be safely written using intrinsics. The optimized code of an iteration over k is:

```
 1: tmp = _mm256_mul_ps( a_0, b_t );
 2: B_0 = _mm256_broadcast_ps( (__m128 *) &B[8] );
 3: d_0 = _mm256_add_ps( d_0, tmp );
 4: tmp = _mm256_mul_ps( a_8, b_t );
 5: b_t = _mm256_shuffle_ps( b_0, b_0, 0x55 );
 6: d_4 = _mm256_add_ps( d_4, tmp );

 7: tmp = _mm256_mul_ps( a_0, b_t );
 8: A_0 = _mm256_load_ps( &A0[8] );
 9: d_1 = _mm256_add_ps( d_1, tmp );
10: tmp = _mm256_mul_ps( a_8, b_t );
11: b_t = _mm256_shuffle_ps( b_0, b_0, 0xaa );
```

```
12: d_5 = _mm256_add_ps( d_5, tmp );

13: tmp = _mm256_mul_ps( a_0, b_t );
14: A_8 = _mm256_load_ps( &A1[8] );
15: d_2 = _mm256_add_ps( d_2, tmp );
16: tmp = _mm256_mul_ps( a_8, b_t );
17: b_t = _mm256_shuffle_ps( b_0, b_0, 0xff );
18: d_6 = _mm256_add_ps( d_6, tmp );

19: tmp = _mm256_mul_ps( a_0, b_t );
20: d_3 = _mm256_add_ps( d_3, tmp );
21: tmp = _mm256_mul_ps( a_8, b_t );
22: b_t = _mm256_shuffle_ps( B_0, B_0, 0x00 );
23: d_7 = _mm256_add_ps( d_7, tmp );
```

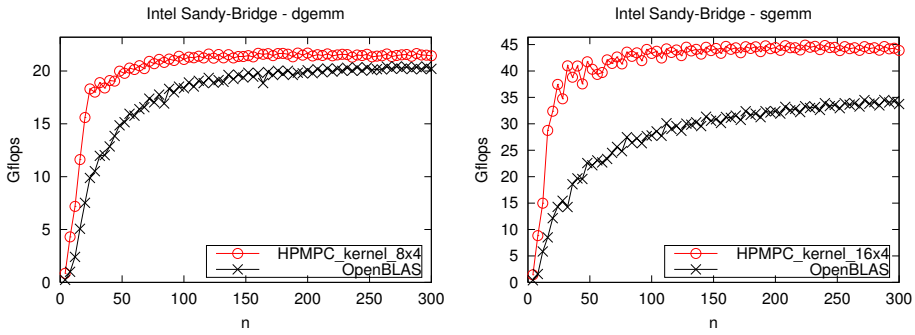
The `broadcast_ps` instructions load 128 bits (4 floats) of B and repeats them on the high and low halves of a 256-bit register. The `shuffle` instructions can shuffle a vector in any combination within the high and low half of a 256-bit register (but in the identical way in the high and low part). Therefore, the combination of the two instructions can be employed to broadcast a B elements to all components of a vector without the need to employ a `broadcast` instruction every clock cycle.

The accumulation registers `d_0` to `d_7` contain the 16×4 sub-matrix of D

$$\texttt{d\_0} \leftarrow \begin{bmatrix} d_{00} \\ d_{10} \\ d_{20} \\ d_{30} \\ d_{40} \\ d_{50} \\ d_{60} \\ d_{70} \end{bmatrix}, \quad \texttt{d\_1} \leftarrow \begin{bmatrix} d_{01} \\ d_{11} \\ d_{21} \\ d_{31} \\ d_{41} \\ d_{51} \\ d_{61} \\ d_{71} \end{bmatrix}, \quad \texttt{d\_2} \leftarrow \begin{bmatrix} d_{02} \\ d_{12} \\ d_{22} \\ d_{32} \\ d_{42} \\ d_{52} \\ d_{62} \\ d_{72} \end{bmatrix}, \quad \texttt{d\_3} \leftarrow \begin{bmatrix} d_{03} \\ d_{13} \\ d_{23} \\ d_{33} \\ d_{43} \\ d_{53} \\ d_{63} \\ d_{73} \end{bmatrix}$$

$$\texttt{d\_4} \leftarrow \begin{bmatrix} d_{80} \\ d_{90} \\ d_{a0} \\ d_{b0} \\ d_{c0} \\ d_{d0} \\ d_{e0} \\ d_{f0} \end{bmatrix}, \quad \texttt{d\_5} \leftarrow \begin{bmatrix} d_{81} \\ d_{91} \\ d_{a1} \\ d_{b1} \\ d_{c1} \\ d_{d1} \\ d_{e1} \\ d_{f1} \end{bmatrix}, \quad \texttt{d\_6} \leftarrow \begin{bmatrix} d_{82} \\ d_{92} \\ d_{a2} \\ d_{b2} \\ d_{c2} \\ d_{d2} \\ d_{e2} \\ d_{f2} \end{bmatrix}, \quad \texttt{d\_7} \leftarrow \begin{bmatrix} d_{83} \\ d_{93} \\ d_{a3} \\ d_{b3} \\ d_{c3} \\ d_{d3} \\ d_{e3} \\ d_{f3} \end{bmatrix}$$

where exadecimal indexes are employ to use a single character in the notation.



(a) Intel Sandy-Bridge dgemm.

(b) Intel Sandy-Bridge sgemm.

Figure 5.3: Performance test of `gemm` kernel for squared matrices of size $n \times n$, $n \in [4, 300]$, on an Intel Sandy-Bridge, code compiled with gcc 4.8.4. Peak performance in double (single) precision is $8 \cdot 2.9 = 23.2$ Gflops ($16 \cdot 2.9 = 46.4$ Gflops).

As in the `dgemm` case, the remaining 8 registers are used for intermediate results, and to prefetch elements from A and B.

5.2.3.3 Results

The test machine is a laptop equipped with the Intel Core i5 2410M processor. The processor has a base frequency of 2.3 GHz and a maximum turbo frequency of 2.9 GHz. During all tests, the processor is found running at the maximum turbo frequency. The processor incorporates 3 MB of L3 cache. In double precision (Figure 5.3a), the proposed `dgemm` routine attains a maximum performance of 21.7 Gflops (93%), that compares with the 20.4 Gflops (88%) of OpenBLAS. In this case, the proposed `dgemm` routine shows a good performance also for small matrix sizes, but the OpenBLAS `dgemm` routine performs rather well. In single precision (Figure 5.3b), the proposed `sgemm` routine attains a maximum performance of 44.9 Gflops (97%), that compares with the 34.5 Gflops (74%) of OpenBLAS. In this case, there is a large performance gap between the proposed routine and the OpenBLAS counterpart, especially for small matrix sizes. The reason is partly due to the fact that the `sgemm` kernel in OpenBLAS makes use of a 8×8 scheme, that in theory optimizes the A and B elements reuse but in practice seems to saturate the shuffle execution unit.

5.2.4 Intel Haswell

The Haswell architecture arrived to the market in 2013 as a "tock" (a new micro-architecture). It is implemented on the same 22 nm FinFET of the Ivy-Bridge architecture, that was a "tick" (shrink to a new process technology) of the Sandy-Bridge micro-architecture. Haswell is found in 4th generation Intel Core processors. The following Broadwell architecture (found in 5th generation Intel Core processors) is a "tick" of Haswell, implementing basically the same micro-architecture on Intel 14 nm FinFET process.

On the ISA side, Haswell presents many improvements. The changes most useful to numerical code are the introduction of the AVX2 ISA (providing 256-bit wide integer SIMD and introducing new FP instructions such as gather and broadcast) and the FMA ISA (containing fused-multiply-accumulate instructions).

The Haswell micro-architecture shares with its predecessor (Sandy-Bridge) the fact that it is a 64-bit superscalar processor that can issue 4 instructions per cycle. It supports two threads per physical core, speculative and out-of-order execution, and register renaming on both GP and FP registers. There are 32 KB of both data and instruction 8-way associative L1 cache per core, and 256 KB of 8-way associative L2 cache per core. There is also a L3 cache shared between CPU cores and GPU.

Among the new micro-architecture features there is the addition of a new execution port, used to execute ALU and branch instructions. The multiplication port in Sandy-Bridge is extended to execute SIMD fused-multiply-accumulate instructions. The addition port present in Sandy-Bridge is also extended to execute a SIMD fused-multiply-accumulate instruction. This means that two 256-bit wide fused-multiply-accumulate can be issued at each clock cyce, doubling the FP theoretical performance with respect to the Sandy-Bridge micro-architecture. In order to sustain the improved FP performance, the bandwidth between registers and L1 cache is doubled, and the Haswell micro-architecture can sustain 2 256-bit loads and 1 256-bit store per clock cycle. The bandwidth L1 and L2 cache is doubled too, and now equal to 64 Byte (256 bit) per clock cycle. In some models it may be present a 4th level of cache.

In both single and double precision, the FMA instruction has a latency of 5 clock cycles (that is identical to the latency of multiplication) and a throughput of 0.5. This means that, in order to fully hide latency of FMA instructions, at least 10 registers have to be used to hold elements of the result sub-matrix C. As a consequence, the `gemm` implementation scheme used in both Core and Sandy-Bridge micro-architectures (where 8 registers are used to hold a sub-matrix of C) can not give more that 80% of full FP throughput.

In the proposed implementation scheme for linear algebra in embedded optimization, the panels from all matrices needs to have the same size (in this case equal to the SIMD width): therefore, it is not possible to use exactly 10 registers. Instead, a new `gemm` implementation scheme is employed, where 12 registers are used to hold elements from the result sub-matrix C, and the remaining 4 registers are used to hold elements from the A and B matrix. In the `gemm` kernel, 3 panels from A and 1 panel from B are streamed at once. The 4 remaining registers are enough to hold 3 elements from the A matrix and 1 element from the B matrix. No additional registers are necessary for intermediate results, thanks to the use of FMA instruction. There are no registers left to prefetch elements from A and B: therefore the 16 FP registers are barely enough to implement this `gemm` scheme.

5.2.4.1 dgemm

Each 256-bit register can hold 4 double-precision FP numbers. In double precision, the fused-multiply-add instructions has a throughput of 2 instruction per clock cycle. Therefore in double precision the full FP throughput of the Haswell architecture is 16 flops per clock cycle.

In the `dgemm` optimization, in the case of the Haswell architecture, at least 10 registers are needed to hide the latency of the fused-multiply-add instruction. Given the constraint that the panel wise has to be the same for all matrices, this means that 12 out of 16 registers must be used to hold a 12×4 sub-matrix of D. The panel height is $b_s = 4$, identical the Sandy-Bridge case.

Similarly to the Sandy-Bridge case, the code can be safely written using intrinsics. The optimized code of an iteration over k is:

```
 1: d_0 = _mm256_fmadd_pd( a_0, b_0, d_0 );
 2: d_4 = _mm256_fmadd_pd( a_4, b_0, d_4 );
 3: d_8 = _mm256_fmadd_pd( a_8, b_0, d_8 );

 4: b_0 = _mm256_shuffle_pd( b_0, b_0, 0x5 );
 5: d_1 = _mm256_fmadd_pd( a_0, b_0, d_1 );
 6: d_5 = _mm256_fmadd_pd( a_4, b_0, d_5 );
 7: d_9 = _mm256_fmadd_pd( a_8, b_0, d_9 );

 8: b_0 = _mm256_permute2f128_pd( b_0, b_0, 0x1 );
 9: d_3 = _mm256_fmadd_pd( a_0, b_0, d_3 );
10: d_7 = _mm256_fmadd_pd( a_4, b_0, d_7 );
```

```
11: d_b = _mm256_fmadd_pd( a_8, b_0, d_b );

12: b_0 = _mm256_shuffle_pd( b_0, b_0, 0x5 );
13: d_2 = _mm256_fmadd_pd( a_0, b_0, d_2 );
14: a_0 = _mm256_load_pd( &A0[4] );
15: d_6 = _mm256_fmadd_pd( a_4, b_0, d_6 );
16: a_4 = _mm256_load_pd( &A1[4] );
17: d_a = _mm256_fmadd_pd( a_8, b_0, d_a );
18: b_0 = _mm256_load_pd( &B[4] );
19: a_8 = _mm256_load_pd( &A2[4] );
```

This scheme is identical the the one employed in the Sandy-Bridge case, with two key differences. Firstly, the multiplication and the addition are fused, and therefore there are no intermediate results. Furthermore, since 12 registers must be employed as accumulation registers, there are only 4 registers left to hold elements from A and B (there is on need to additional registers for intermediate results). These 4 registers can barely hold the 3 elements of A and the element of B that are reused as arguments of the fused-multiply-add. There are no extra registers available to aggressively prefetch A and B elements. Therefore this implementation scheme must heavily rely on the advanced core features such as out-of-order computation, register renaming and hardware prefetch.

5.2.4.2 sgemm

Each 256-bit register can hold 8 single-precision FP numbers. In single precision, the fused-multiply-add instructions has a throughput of 2 instruction per clock cycle. Therefore in single precision the full FP throughput of the Haswell architecture is 32 flops per clock cycle.

Also in the `sgemm` optimization, in the case of the Haswell architecture, at least 10 registers are needed to hide the latency of the fused-multiply-add instruction. Given the constraint that the panel wise has to be the same for all matrices, this means that 12 out of 16 registers must be used to hold a 24×4 sub-matrix of D. The panel height is $b_s = 8$, identical the Sandy-Bridge case.

Similarly to the Sandy-Bridge case, the code can be safely written using intrinsics. The optimized code of an iteration over k is:

```
1: b_0 = _mm256_broadcast_ps( (__m128 *) &B[0] );
2: d_0 = _mm256_fmadd_ps( a_0, b_0, d_0 );
3: d_4 = _mm256_fmadd_ps( a_8, b_0, d_4 );
```

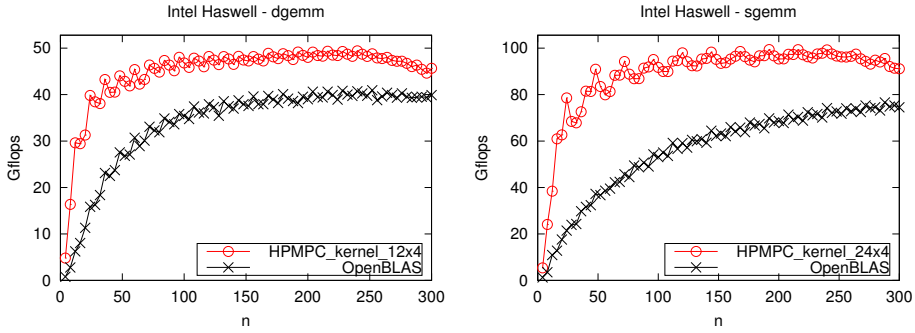
```
 4: d_8 = _mm256_fmadd_ps( a_g, b_0, d_8 );

 5: b_0 = _mm256_permute_ps( b_0, 0xb1 );
 6: d_1 = _mm256_fmadd_ps( a_0, b_0, d_1 );
 7: d_5 = _mm256_fmadd_ps( a_8, b_0, d_5 );
 8: d_9 = _mm256_fmadd_ps( a_g, b_0, d_9 );

 9: b_0 = _mm256_permute_ps( b_0, 0x4e );
10: d_2 = _mm256_fmadd_ps( a_0, b_0, d_2 );
11: d_6 = _mm256_fmadd_ps( a_8, b_0, d_6 );
12: d_a = _mm256_fmadd_ps( a_g, b_0, d_a );

13: b_0 = _mm256_permute_ps( b_0, 0xb1 );
14: d_3 = _mm256_fmadd_ps( a_0, b_0, d_3 );
15: a_0 = _mm256_load_ps( &A0[8] );
16: d_7 = _mm256_fmadd_ps( a_8, b_0, d_7 );
17: a_8 = _mm256_load_ps( &A1[8] );
18: d_b = _mm256_fmadd_ps( a_g, b_0, d_b );
19: a_g = _mm256_load_ps( &A2[8] );
```

Again, this scheme is identical the the one employed in the Sandy-Bridge case, with the same two key differences that in the double precision case. There are no extra registers available to aggressively prefetch A and B elements. Therefore this implementation scheme must heavily rely on the advanced core features such as out-of-order computation, register renaming and hardware prefetch.

5.2.4.3 Results

The test machine is a laptop equipped with the Intel Core i7 4800MQ processor, that has a base frequency of 2.7 GHz and a maximum turbo frequency of 3.7 GHz, and it is equipped with 6 MB L3 cache. If AVX or AVX2 code is employed, the turbo frequency is lowered to 3.3 GHz, as it is in these tests. At 3.3 GHz, the full FP throughput per core is of 52.8 Gflops in double precision and of 105.6 Gflops in single precision. In double precision (Figure 5.4a) the proposed `dgemm` routine reaches 49.4 Gflops (93.6%), while in single precision (Figure 5.4b) the proposed `sgemm` routine reaches 99.4 Gflops (94.1%). As a comparison, the OpenBLAS counterparts reach 40.9 Gflops (77.5%) and 76.6 Gflops (72.5%). Even more important than absolute performance, the proposed implementation scheme gives much better performance for small matrices, especially in single precision.

As a note on the implementation of the `potrf` and `trtri` kernels, there the



(a) Intel Haswell dgemm.

(b) Intel Haswell sgemm.

Figure 5.4: Performance test of `gemm` kernel for squared matrices of size $n \times n$, $n \in [4, 300]$, on an Intel Haswell processor, code compiled with gcc 4.9.2. Peak performance in double (single) precision is $16 \cdot 3.3 = 52.8$ Gflops ($32 \cdot 3.3 = 105.6$ Gflops).

`fnmadd` intrinsic is used in place of the `fmadd` intrinsic in the kernel loop. The `gcc` compiler emits a combination of `fmadd` and `xor` instructions in place of the `fnmadd` instructions, decreasing performance by about 20% compared to the `gemm` kernel. This is solved by using a customized version of the `gcc` compiler, as described in Appendix A.

5.2.5 Intel Skylake

The Skylake architecture arrived to the market in late 2015 as a "tock" (a new micro-architecture). It is implemented on the same 14 nm FinFET of the Broadwell architecture, that was a "tick" (shrink to a new process technology) of the Haswell micro-architecture. Skylake is found in 6th generation Intel Core processors. Skylake will be followed by another "tock", called Kaby Lake (that will be found in the 7th generation Intel Core processors), expected in 2016, that will be implemented in the same 14 nm FinFET process. At the time of writing, there are no details about the new architecture, even if some speculations suggest the possible widespread introduction of the AVX512 ISA, that is present only on high-end Skylake Xeon processors. The reason for this double "tock" is probably due to the difficulties in developing the 10 nm process technology. The "tick" of Skylake should be called Cannonlake, and due sometimes in 2017.

On the ISA side, Skylake does not presents many improvements, and in partic-

ular there are no new ISAs improving FP code.

At the time of writing, there are not many details about the Skylake micro-architecture. It is a 64-bit superscalar processor, and it appears that it can issue 5 or 6 instructions per cycle. It supports two threads per physical core, speculative and out-of-order execution, and register renaming on both GP and FP registers. There are 32 KB of both data and instruction 8-way associative L1 cache per core, and 256 KB of 4-way associative L2 cache per core. There is also a L3 cache shared between CPU cores and GPU. In some models it may be present a 4th level of cache.

A feature affecting the `gemm` implementation scheme is that in both single and double precision, the FMA instruction has a latency of 4 clock cycles (down from 5 cycles in the Haswell micro-architecture) and a throughput of 0.5. This means that, in order to fully hide latency of FMA instructions, 8 accumulation registers (instead of 10) should be enough to obtain full throughput. As a consequence, a `gemm` implementation scheme where 8 registers are used to hold a sub-matrix of C should give full FP throughput in the case of the Skylake micro-architecture. This should slightly boost performance for small matrices. Compared to the implementation scheme for Haswell, this should leave 4 registers free for more aggressive prefetch of data, likely slightly improving peak performance too. Furthermore, since Broadwell there is a radix-1024 divider: this reduces the latency of divisions and square-roots instructions, further improving performance e.g. in the factorization of small matrices.

The above speculations about the performance of a `gemm` implementation scheme and of factorization routines could not be tested on a Skylake processor yet.

Note: at the time of printing the final version of the thesis, a Skylake processor has been quickly evaluated. A `gemm` kernel using 8 accumulation registers is found to deliver about 80% of full throughput: even if this value is slightly higher than in the Haswell case, the Haswell scheme using 12 accumulation registers has to be employed also in the Skylake case in order to obtain full throughput.

5.2.6 AMD K10

The K10, or 10h, is a micro-architecture designed by AMD. The first products based on this micro-architecture were released in 2007.

The K10 micro-architecture is an evolutionary step of the K8 micro-architecture. It is a 64-bit superscalar processor that can issue 3 instructions per cycle. It

supports speculative and out-of-order execution, and register renaming. There are 64 KB 2-way associative L1 instruction cache and data cache, plus 512 KB 16-way associative exclusive L2 cache per core. Additionally, there are 2 MB L3 cache shared among cores. The vector units width has been increased from 64-bit in the K8 to 128-bit in the K10 micro-architecture.

On the ISA side, it supports SSEx ISAs up to SSE3, plus the SSE4a ISA (not supported by Intel architectures). For the purpose of linear algebra routines implementation, the SSE, SSE2 and SSE3 ISAs are considered, that are common to Intel architectures. Therefore, the `gemm` kernels developed for the Intel Core micro-architecture works also for the AMD K10 micro-architecture.

However, an important micro-architectural difference between Intel Core and AMD K10 is that in the latter the shuffles (both FP and universal) are performed on the multiplication unit. Therefore, shuffle instructions compete with multiplications for the same execution unit, reducing throughput. As a solution, in double precision the `movddup` instruction can be used to broadcast a double-precision FP number to both the lower and the upper half of a 128-bit register, without the need to use shuffle instructions. Since loads are performed on a different execution unit, this allows the multiplication instructions to reach full throughput.

In single precision, this scheme can not be employed, since there is not an equivalent to the `movddup` instruction until the introduction of the AVX instruction set. In the optimized `sgemm` in OpenBLAS, this is solved by storing a sub-matrix of B in a memory buffer such that each B element is repeated 4 times, and therefore the 4 copies of the same element can be loaded on all vector components by means of a simple load instruction. The buffer is reused in the product with a large sub-matrix of A, therefore well hiding the cost of building the B buffer. However, this implementation scheme can not be employed in the proposed implementation scheme for linear algebra in embedded optimization, since the matrices are not copied into buffers on-line. Therefore, the 8×4 `sgemm` kernel developed for the Intel Core micro-architecture is employed (even if sub-optimal) also for the AMD K10 architecture.

5.2.6.1 dgemm

The 4×4 `dgemm` kernel tailored to the AMD K10 micro-architecture is analogue to the corresponding kernel developed for the Intel Core, with the difference that shuffle instructions are replaced by `movddup` instructions. Furthermore, the code is implemented in C code with intrinsics, that gives good performance thanks to the out-of-order and register-renaming capabilities of the micro-architecture.

```
a_0 = _mm_load_pd(&A[0]);
a_2 = _mm_load_pd(&A[2]);

b_0 = _mm_loaddup_pd(&B[0]);
b_1_0 = b_0;
b_0 = _mm_mul_pd( a_0, b_0 );
d_0 = _mm_add_pd( d_0, b_0 );
b_1 = _mm_mul_pd( a_2, b_1 );
d_4 = _mm_add_pd( d_4, b_1 );

b_0 = _mm_loaddup_pd(&B[1]);
b_1 = b_0;
b_0 = _mm_mul_pd( a_0, b_0 );
d_1 = _mm_add_pd( d_1, b_0 );
b_1 = _mm_mul_pd( a_2, b_1 );
d_5 = _mm_add_pd( d_5, b_1 );

b_0 = _mm_loaddup_pd(&B[2]);
b_1 = b_0;
b_0 = _mm_mul_pd( a_0, b_0 );
d_2 = _mm_add_pd( d_2, b_0 );
b_1 = _mm_mul_pd( a_2, b_1 );
d_6 = _mm_add_pd( d_6, b_1 );

b_0 = _mm_loaddup_pd(&B[3]);
b_1 = b_0;
b_0 = _mm_mul_pd( a_0, b_0 );
d_3 = _mm_add_pd( d_3, b_0 );
b_1 = _mm_mul_pd( a_2, b_1 );
d_7 = _mm_add_pd( d_7, b_1 );
```

5.3 ARMv7A

The ARMv7A is a 32-bit RISC architecture. As most RISC architectures, it is a load/store architecture (i.e. only load and store instructions can access memory, all other instructions operate on registers) and it features simple addressing modes. The ARMv7 architecture is divided into 3 profiles: Application (A), Real-time (R) and Micro-controller (M) profiles. The A profile is intended for the highest performance applications, it employs a Memory Management Unit (MMU), and it is influenced by multi-tasking OS requirements.

The architecture defines 15 32-bit GP registers. In ARMv7A, there are two ISA: ARM (32-bit encoding) and Thumb-2 (mixed of 16- and 32-bit encoding, for higher code density). Regarding FP computation, there are two ISAs: VFP (VFPv3 or VFPv4 in ARMv7A) and NEON (NEON or NEONv2 in ARMv7A). The VFP ISA is essentially a scalar ISA, processing both single and double FP numbers, while NEON is a SIMD one that can process many types of vector integers and single-precision FP numbers. In most implementations of VFP an in all implementations of NEON, there are 32 64-bit FP registers. Instructions usually take 3 operands, and thus none of the input operands needs to the overwritten.

Since version 0.2.9, OpenBLAS supports ARMv7A, but using only the scalar VFP unit. The NEON code presented in the following is product of original research and it outperforms the implementation in OpenBLAS.

5.3.1 ARM Cortex A9

The Cortex A9 is a 32-bit processor designed by ARM, on the market since 2010. At that time, it was the higher performing processor by ARM. Compared to its predecessor, the Cortex A8, the Cortex A9 adds out-of-order capabilities and a multi-core design. On the FP side, Cortex A9 greatly improves performance compared to Cortex A8: in fact, the VFP unit in the Cortex A8 is not pipelined, and therefore many instructions are about an order of magnitude slower than in the Cortex A9. The Cortex A9 can be found in the SoC equipping many smartphones and tablets of a few years ago.

The Cortex A9 is a multicore design with up to 4 coherent cores. The Cortex A9 micro-architecture is superscalar with an issue capability of two instructions per cycle (even if not all combinations of instructions can be co-issued, e.g. numerical experiments shows that it can not co-issue floating-point instructions and load instructions). It performs speculative and partially out-of-order execution and register renaming in the GP registers (but not on the FP registers). There can be 4-way associative 16, 32 or 64 KB of both data and instruction L1 cache per core. There may be an external L2 cache shared among cores, with size ranging from 128 KB to 8 MB. The cache line size is 32 byte.

The Cortex A9 can have two floating-point units: VFPv3 and NEON. The former is a scalar unit that can process both single and double precision FP numbers, and it is fully IEEE compilant. The latter is a vector unit that can process small vectors of 4 single-precision FP numbers, but does not provide support to double-precision FP numbers, nor is fully IEEE compilant, since it only support the round-to-nearest mode. In the Cortex A9, both FP units are

optional. However, in most cases both of them are present, since they are used in processing multimedia. The FP datapath is 64-bit wide.

5.3.1.1 dgemm

The vector NEON instruction set does not support double-precision floating-point numbers. As a consequence, the scalar VFP instruction set has to be used.

The Cortex A9 implements the VFPv3 version, that has a multiply-accumulate instruction (MLA) but not a fused-multiply-accumulate instruction (FMA). The difference between the two instructions is that in the multiply-accumulate instruction the result is rounded twice, once after the multiplication and once after the addition, while in the fused-multiply-accumulate the result is rounded only once, after the final addition: the FMA is thus more accurate than MLA. Therefore, the FMA is more accurate than MLA or than the sequence of multiplication and addition instructions.

The VFP unit has 32 scalar double-precision registers. Out of them, 16 can be used as accumulation registers, to hold a 4×4 sub-matrix of C. Thus a 4×4 `dgemm` kernel is used, with panel high $b_s = 4$. Of the remaining 16 registers, 8 are enough to hold a 4 elements from the A matrix a 4 elements from the B matrix. This means that there are still 8 registers available to more aggressively prefetch the following 4 elements from the A and B matrices.

The optimized code of an iteration over k is

```
 1: fmacd  d0, d16, d20
 2: fmacd  d1, d17, d20
 3: fmacd  d2, d18, d20
 4: fmacd  d3, d19, d20
 5: fldd   d20, [r1, #64]

 6: fmacd  d4, d16, d21
 7: fmacd  d5, d17, d21
 8: fmacd  d6, d18, d21
 9: fmacd  d7, d19, d21
10: fldd   d21, [r1, #72]

11: fmacd  d8, d16, d22
12: fmacd  d9, d17, d22
```

```
13: fmacd  d10, d18, d22
14: fmacd  d11, d19, d22
15: fldd   d22, [r1, #80]

16: fmacd  d12, d16, d23
17: fldd   d16, [r0, #64]
18: fmacd  d13, d17, d23
19: fldd   d17, [r0, #72]
20: fmacd  d14, d18, d23
21: fldd   d18, [r0, #80]
22: fmacd  d15, d19, d23
23: fldd   d23, [r1, #88]
24: fldd   d19, [r0, #88]
```

In the GCC inline assembly, `fmacd` is the FP multiply-accumulate instruction (first argument is the accumulation register, the second and the third are the multiplication factors), and `fldd` the FP load, both operating on double-precision FP numbers. The Cortex A9 can issue a multiply-accumulate instruction every other cycle: thus by interleaving multiply-accumulate instructions with load instructions, the load instruction can be performed in the idle cycle. On average less than one instruction is issued every cycle, that is below the maximum issue capability of 2 instructions per cycle.

In the above code, 64-bit registers d0-d15 are used to hold a 4×4 sub-matrix of C:

$$\begin{bmatrix} \mathrm{d0} & \mathrm{d4} & \mathrm{d8} & \mathrm{d12} \\ \mathrm{d1} & \mathrm{d5} & \mathrm{d9} & \mathrm{d13} \\ \mathrm{d2} & \mathrm{d6} & \mathrm{d10} & \mathrm{d14} \\ \mathrm{d3} & \mathrm{d7} & \mathrm{d11} & \mathrm{d15} \end{bmatrix} \leftarrow \begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{bmatrix}$$

while registers d16-d23 are used to hold vectors from A and B matrices:

$$\begin{bmatrix} \mathrm{d16} \\ \mathrm{d17} \\ \mathrm{d18} \\ \mathrm{d19} \end{bmatrix} \leftarrow \begin{bmatrix} a_{0k} \\ a_{1k} \\ a_{2k} \\ a_{3k} \end{bmatrix}, \quad \begin{bmatrix} \mathrm{d20} \\ \mathrm{d21} \\ \mathrm{d22} \\ \mathrm{d23} \end{bmatrix} \leftarrow \begin{bmatrix} b_{0k} \\ b_{1k} \\ b_{2k} \\ b_{3k} \end{bmatrix},$$

while at the following iteration over k the registers d24-d31 are used instead.

A performance test shows that the best performance is obtained for data fitting L1 cache: this hints at the fact that hardware prefetch is not implemented. If software prefetch is employed, the high-performance is attained also for data fitting in L2 cache.

The large number of registers and the multiply-accumulate instruction makes the optimization for this kernel easy.

5.3.1.2 sgemm

In single-precision, it is possible to choose between two FP units: VFP and NEON. The former is a scalar unit (despite the name) and is fully IEEE compilant; the latter is a vector unit that can perform some integer and floating-point instruction (but e.g. no divisions nor square root) but it is not fully IEEE compilant (e.g. it only supports round-to-nearest and always flushes denormals to zero). This is not an issue in case of MPC, thus the higher-performing vector NEON unit is preferred.

The NEON instruction set supports a set of 32 double-word (or 64-bit) registers (d0-d31), that can be seen as 16 quad-word (or 128-bit) registers (q0-q15). The lower 16 double-word registers can be seen as 32 single-word (or 32-bit) registers (s0-s31) and accessed individually. All instructions are flexible and can operate on vectors of 1, 2 or 4 packed single-precision FP numbers, corresponding to the s, d, or q registers. This means e.g. that it is possible to operate on the high double-word of a quad-word register, or on all single words of a quad-word register in the set q0-q7. This flexibility makes the code optimization much easier than in the x86 architecture, where only the lower element or the whole vector can be accessed and shuffle and permute instructions must be heavily employed.

In the implementation of the `sgemm` kernel, out of the 16 quad-word registers, 8 can be used as accumulation registers to hold a 8×4 sub-matrix of C. Thus a 8×4 kernel is used, with panel height $b_s = 4$. Of the remaining 8 registers, 3 are enough to hold two 4-wide vectors from matrix A and 4-wide vector from matrix B needed at an iteration over k, and thus 6 registers can be used to aggressively prefetch the vectors from A and B needed for two consecutive k iterations.

The NEON version implemented in the Cortex A9 supports multiply-accumulate instruction, but not fused-multiply-accumulate instruction. The multiply-accumulate instruction is given in two variants, a vector-times-vector one (where all elements of two registers are multiplied element-wise, and then added element-wise to the accumulation register), and a vector-times-scalar one (where all elements of a register are multiplied by a single element on a scalar register, and then added element-wise to all elements of the accumulation register). This instruction is rather powerful, and can be seen as the combination of a shuffle, a multiplication and an addition in the x86 SSE instruction set.

In the Cortex A9, the multiply-accumulate instruction has a throughput of 2, and thus the theoretical peak performance is of 4 flops per cycle (a fused multiply-add every other cycle). However, in practice the attained performance in the `sgemm` kernel is rather lower. In fact, despite the instruction issue capability of 2 instructions per cycle, the Cortex A9 seems to have a single port for FP and load/store instructions, and therefore these instructions can not be co-issued. Even worse, it is well known that, due to the structure of the FP pipeline, there is a performance penalty when mixing VFP and NEON instructions. The numerical tests performed during the optimization of the `sgemm` kernel shows that there is a similar performance penalty in mixing multiply-accumulate instructions with load instructions. This further limits the attainable performance, since load instructions can not be issued in the idle cycle between two multiply-accumulate. The best performing code performs keeps the multiply-accumulate instructions and the load instructions as separated as possible.

The optimized code of two following iterations over k is

```
 1: vmla.f32  q8, q2, d0[0]
 2: vmla.f32  q9, q2, d0[1]
 3: vmla.f32  q10, q2, d1[0]
 4: vmla.f32  q11, q2, d1[1]

 5: vmla.f32  q12, q4, d0[0]
 6: vmla.f32  q12, q4, d0[1]
 7: vmla.f32  q14, q4, d1[0]
 8: vmla.f32  q15, q4, d1[1]

 9: vmla.f32  q8, q3, d2[0]
10: vmla.f32  q9, q3, d2[1]
11: vmla.f32  q10, q3, d3[0]
12: vmla.f32  q11, q3, d3[1]

13: vmla.f32  q12, q5, d2[0]
14: vmla.f32  q13, q5, d2[1]
15: vmla.f32  q14, q5, d3[0]
16: vmla.f32  q15, q5, d3[1]

17: vld1.64   {d0, d1, d2, d3},   [r2:128]!
19: vld1.64   {d4, d5, d6, d7},   [r0:128]!
18: vld1.64   {d8, d9, d10, d11}, [r1:128]!
```

The `vmla.f32` is the NEON multiply-accumulate operating on 32 bit FP num-

bers. The first argument is the accumulation register, the second and the third are the multiplication factors. The first 2 arguments are quad-word registers, the third is the scalar value broadcast in the multiplication. The `vld1.64` instruction is the NEON load of contiguous 64-bit values: it can load up to 4 64-bit registers (specified in the curly brackets), from the memory location starting at the value in the GP register in squared brackets together with an optional alignment hint (in this case 128 bit alignment); the exclamation mark is used to increase the value of the GP register of the right quantity after the load.

The quad-word registers `q8` to `q15` are used to hold a 8×4 sub-matrix of D, as

$$\texttt{q8} \leftarrow \begin{bmatrix} d_{00} \\ d_{10} \\ d_{20} \\ d_{30} \end{bmatrix}, \quad \texttt{q9} \leftarrow \begin{bmatrix} d_{01} \\ d_{11} \\ d_{21} \\ d_{31} \end{bmatrix}, \quad \texttt{q10} \leftarrow \begin{bmatrix} d_{02} \\ d_{12} \\ d_{22} \\ d_{32} \end{bmatrix}, \quad \texttt{q11} \leftarrow \begin{bmatrix} d_{03} \\ d_{13} \\ d_{23} \\ d_{33} \end{bmatrix}$$

$$\texttt{q12} \leftarrow \begin{bmatrix} d_{40} \\ d_{50} \\ d_{60} \\ d_{70} \end{bmatrix}, \quad \texttt{q13} \leftarrow \begin{bmatrix} d_{41} \\ d_{51} \\ d_{61} \\ d_{71} \end{bmatrix}, \quad \texttt{q14} \leftarrow \begin{bmatrix} d_{42} \\ d_{52} \\ d_{62} \\ d_{72} \end{bmatrix}, \quad \texttt{q15} \leftarrow \begin{bmatrix} d_{43} \\ d_{53} \\ d_{63} \\ d_{73} \end{bmatrix}$$

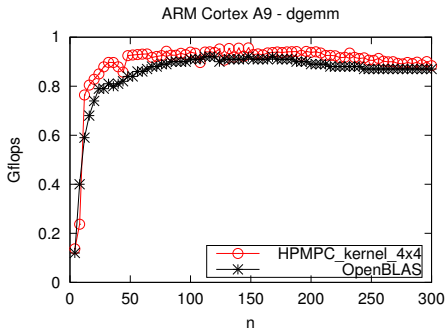
The quad-word registers `q0` to `q5` are used to hold elements of A and B for two iterations, as

$$\texttt{q0} \leftarrow \begin{bmatrix} b_{0,k+0} \\ b_{1,k+0} \\ b_{2,k+0} \\ b_{3,k+0} \end{bmatrix}, \quad \texttt{q1} \leftarrow \begin{bmatrix} b_{0,k+1} \\ b_{1,k+1} \\ b_{2,k+1} \\ b_{3,k+1} \end{bmatrix}, \quad \begin{matrix} \texttt{q2} \leftarrow \begin{bmatrix} a_{0,k+0} \\ a_{1,k+0} \\ a_{2,k+0} \\ a_{3,k+0} \end{bmatrix} \\ \texttt{q3} \leftarrow \begin{bmatrix} a_{0,k+1} \\ a_{1,k+1} \\ a_{2,k+1} \\ a_{3,k+1} \end{bmatrix} \end{matrix}, \quad \begin{matrix} \texttt{q4} \leftarrow \begin{bmatrix} a_{4,k+0} \\ a_{5,k+0} \\ a_{6,k+0} \\ a_{7,k+0} \end{bmatrix} \\ \texttt{q5} \leftarrow \begin{bmatrix} a_{4,k+1} \\ a_{5,k+1} \\ a_{6,k+1} \\ a_{7,k+1} \end{bmatrix} \end{matrix}$$

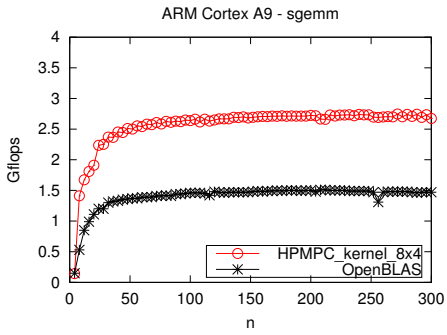
The quad-word registers `q6` and `q7` are not employed.

5.3.1.3 Results

The test machine is a development board called Wandboard Quad. The processor is the Freescale i.MX6 SoC, that has a quad-core Cortex A9 running at 1 GHz and 1 MB L2 cache. In this thesis, only single-thread code is considered. As shown in Figure 5.5, the `dgemm` and `sgemm` kernels attain a best performance respectively of 0.95 Gflops (95% of full FP throughput) and 2.74 Glops (68%). As a comparison, OpenBLAS attains a best performance respectively of 0.92 Gflops (92%) and 1.51 Gflops (38%). In single precision, the much lower performance of OpenBLAS is mainly due to the fact that it does not make use of



(a) Cortex A9 dgemm.



(b) Cortex A9 sgemm.

Figure 5.5: Performance test of `gemm` kernel for squared matrices of size $n \times n$, $n \in [4, 300]$, on an ARM Cortex A9, code compiled with gcc 4.6.3. Peak performance in double (single) precision is $1 \cdot 1 = 1$ Gflops ($4 \cdot 1 = 4$ Gflops).

the vector NEON unit (since in Cortex A9 it is not fully IEEE compilant) but rather of the scalar VFP unit.

5.3.2 ARM Cortex A15

The Cortex A15 is the highest performing 32-bit processor designed by ARM. Originally designed for use in servers, it is present on the market since 2012. It can use up to 1 TB of memory thanks to the 40-bit Large Physical Address Extensions. It can be found in high-end smartphones and tables or in laptops, either alone or in big.LITTLE configuration with lower-power Cortex A7.

The Cortex A15 is a multicore design, with up to 4 coherent cores per cluster, and up to 4 clusters per physical chip. The Cortex A15 micro-architecture is superscalar with an issue capability of 3 instructions per clock cycle (and, as an improvement over the Cortex A9, the Cortex A15 can co-issue FP instructions and load instructions). It supports speculative and out-of-order execution, and register renaming on both GP and FP registers. There can be up to 4 cores per cluster, with 2-way associative 32 KB of L1 data and instruction caches, and up to 4 MB of integrated L2 cache per cluster. The cache line size is 64 byte, twice as much as the Cortex A9.

The Cortex A15 has two FP units per core: VFPv4 and NEONv2. The main

difference with respect to the VFPv3 and NEON units present in older Cortex A9 is the presence of fused-multiply-accumulate instructions.

5.3.2.1 dgemm

In double-precision, the scalar VFPv4 units has to be used. The proposed code is the same as the Cortex A9 (with the difference that half the prefetch instructions are needed, since the cache line size is the double than in the Cortex A9), even if a version using the fused-multiply-accumulate may be written.

Compared to Cortex A9, in the Cortex A15 the multiply-accumulate instruction has a throughput of 1. Furthermore, a FP instruction and a load instruction can be co-issued in the same clock cycle: thus the full (and actual) FP throughput is doubled compared to the Cortex A9. The FP datapath is 128-bit wide, doubled compared to Cortex A15.

5.3.2.2 sgemm

Also in the case of Cortex A15, in single-precision it is possible to choose between VFP and NEON units. Since the latter is higher performing, a version of the `sgemm` kernel using NEON instruction will be presented.

In Cortex A15, the NEON instruction set is implemented in the NEONv2 version, that has also the more accurate fused-multiply-accumulate instruction. However, this is implemented only in the vector-times-vector form, and tuning experiments shows that it has a lower throughput compared to the multiply-accumulate instruction. Therefore the latter is chosen, since it has an higher throughput (1 instruction per clock cycle), and is implemented also in the vector-times-scalar form.

In the Cortex A15, there is a set of 16 quad-word registers (q0-15), that can be seen as a set of 32 double-word registers (d0-d31). As an improvement over Cortex A9, in the Cortex A15 there are no penalty issues in mixing VFP and NEON instructions, nor in mixing NEON and load instructions. Furthermore, tuning experiments show that it can co-issue a NEON multiply-accumulate and a VFP load, but it can not co-issue a NEON multiply-accumulate and a NEON load. The best performing code is thus obtained by interleaving a NEON multiply-accumulate and a VFP load.

The fact that there are no penalty issues in mixing multiply-accumulate and

load instructions implies that it is possible to use even more registers to hold a sub-matrix of C, reducing in this way the number of memory operations. In particular, a 12×4 kernel is found having better performance than a 8×4 kernel. In the 12×4 kernel, 12 quad-word registers (q4-q15) are used as accumulation register to hold a 12×4 sub-matrix of C, while the 4 remaining registers (q0-q3) are used to hold three 4-wide vectors from the A matrix and one from the B matrix.

The optimized code of two following iterations over k is

```
 1: vmla.f32  q4, q1, d0[0]
 2: vldr      d6, [r2, #0]
 3: vmla.f32  q5, q1, d0[1]
 4: vldr      d7, [r2, #8]
 5: vmla.f32  q6, q1, d1[0]
 6: vmla.f32  q7, q1, d1[1]
 7: vldr      d2, [r0, #0]

 8: vmla.f32  q8, q2, d0[0]
 9: vldr      d3, [r0, #8]
10: vmla.f32  q9, q2, d0[1]
11: vmla.f32  q10, q2, d1[0]
12: vmla.f32  q11, q2, d1[1]
13: vldr      d4, [r3, #0]

14: vmla.f32  q12, q3, d0[0]
15: vldr      d5, [r3, #8]
16: vmla.f32  q13, q3, d0[1]
17: vldr      d0, [r1, #0]
18: vmla.f32  q14, q3, d1[0]
19: vmla.f32  q15, q3, d1[1]
20: vldr      d1, [r1, #8]

21: vmla.f32  q4, q1, d4[0]
22: vldr      d6, [r2, #16]
23: vmla.f32  q5, q1, d4[1]
24: vldr      d7, [r2, #24]
25: vmla.f32  q6, q1, d5[0]
26: vmla.f32  q7, q1, d5[1]
27: vldr      d2, [%2, #16]

28: vmla.f32  q8, q0, d4[0]
29: vldr      d3, [%2, #24]
```

```
30: vmla.f32  q9, q0, d4[1]
31: vmla.f32  q10, q0, d5[0]
32: vmla.f32  q11, q0, d5[1]
33: vldr      d0, [%5, #16]

34: vmla.f32  q12, q3, d4[0]
35: vldr      d1, [%5, #24]
36: vmla.f32  q13, q3, d4[1]
37: vldr      d4, [%3, #16]
38: vmla.f32  q14, q3, d5[0]
39: vmla.f32  q15, q3, d5[1]
40: vldr      d5, [%3, #24]
```

with lines 1-20 corresponding to the first iteration over k and lines 21-40 corresponding to the second iteration over k. The instruction `vmla.f32` is the NEON 4-wide vector multiply-accumulate of single-precision FP numbers, and the instruction `vldr` is the VFP load. Notice that the role of registers `q0` (i.e. d0-d1) and `q2` (i.e. d4-d5) is inverted in the second iteration, compared to the first one: this choice is made to separate as much as possible the loading of registers and their following use.

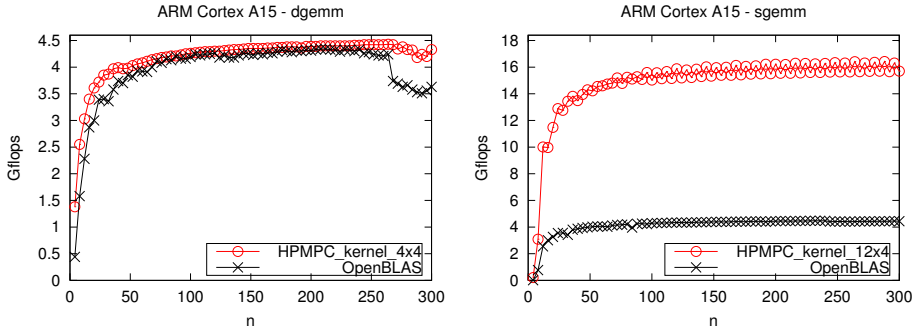
5.3.2.3 Results

The test processor is one core of the NVIDIA Tegra K1 SoC, running at 2.3 GHz. As shown in Figure 5.6, the dgemm and sgemm kernels attain a best performance respectively of 4.41 Gflops (95.8% of full FP throughput) and 16.33 Gflops (88.7%). Similarly to the Cortex A9 case, in double precision OpenBLAS gives a similar performance, but in single precision it is much slower, since it does not make use of the vector unit NEON.

5.3.3 ARM Cortex A7

The Cortex A7 is a low-power 32-bit processor designed by ARM. Present on the market since 2013, it can be found alone in low-end smartphones and tablets, or in combination with the Cortex A15 (big.LITTLE technology) in high-end devices. It is fully feature compatible with Cortex A15, but the design focus in on low-power consumption instead of high-performance.

The Cortex A7 micro-architecture is partially superscalar, being able to double-issue only few combinations of instructions. It supports in-order execution,



(a) Cortex A15 dgemm. **(b)** Cortex A15 sgemm.

Figure 5.6: Performance test of `gemm` kernel for squared matrices of size $n \times n$, $n \in [4, 300]$, on an ARM Cortex A15, code compiled with gcc 4.?. Peak performance in double (single) precision is $2 \cdot 2.3 = 4.6$ Gflops ($8 \cdot 2.3 = 18.4$ Gflops).

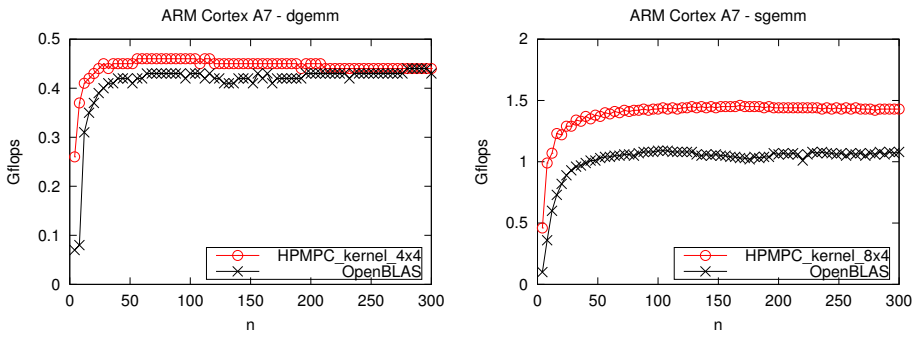
without any register renaming. There can be up to 8 cache-coherent cores per cluster, with 32 KB of both instruction and data L1 cache per core, and up to 1 MB integrated L2 cache. The cache line size is 64 byte for L1 data and L2 caches, and 32 byte for L1 instruction cache. It has the VFPv4 and NEONv2 FP units, and the FP datapath is 64 bit (same as Cortex A9).

5.3.3.1 dgemm

In double precision, the code for both `dgemm` is exactly the same as for Cortex A15. However, Cortex A7 can only perform a double-precision MLA every 4 cycles: as a consequence the performance-per-cycle is half of Cortex A9 and a quarter of Cortex A15, with a full FP throughput of 0.5 flops per cycle.

5.3.3.2 sgemm

In single precision, the Cortex A7 can perform a VFP MLA every cycle, or a NEON 4-wide MLA every 4 cycles (NEON can effectively execute 32-bit per cycle): so the full FP throughput is the same for VFP and NEON, namely 2 flops per cycle. However, the Cortex A7 shows the same performance penalties as the Cortex A9, and the penalty in mixing FP loads and MLA apply to both VFP and NEON. In practice, using NEON it is possible to have a slightly better



(a) Cortex A7 dgemm.

(b) Cortex A7 sgemm.

Figure 5.7: Performance test of `gemm` kernel for squared matrices of size $n \times n$, $n \in [4, 300]$, on an ARM Cortex A7, code compiled with gcc 4.?. Peak performance in double (single) precision is $0.5 \cdot 1 = 0.5$ Gflops ($2 \cdot 1 = 2$ Gflops).

performance, so the `sgemm` kernel for Cortex A7 is the same as for Cortex A9, with the difference that it uses half the prefetch instructions (being the cache line twice as long).

5.3.3.3 Results

Our test machine is Cubieboard 2, a development board equipped with the Allwinner A20 SoC: the CPU is a dual-core Cortex A7 @ 1.0 GHz, with 512 KB of L2 cache. The full FP throughput is equal to 0.5 Gflops in double precision and 2 Gflops in single precision. The proposed `dgemm` routine (Figure 5.7a) reaches the 92% of the full FP throughput, while the `sgemm` routine (Figure 5.7b) reaches the 72%. The `dgemm` routine in OpenBLAS has similar performance, while the `sgemm` routine performs slightly worse: again, this is mainly due to the use of the VFP unit in place of the NEON one.

5.4 ARMv8A

The ARMv8A is a 64-bit RISC architecture. It is the 64-bit version of the ARMv7A ISA, and it introduces several enhancements.

There are two main execution states: AArch64 and AArch32. The former supports the ARMv8A ISA, while the latter supports the old ARM and Thumb ISA in ARMv7A. Therefore, ARMv8A processor can execute ARMv7A code in AArch32 mode.

The ARMv8A ISA supports 31 64-bit GP registers, plus an additional register that acts as zero register. The registers are called 'X' registers, and the lower half of them is accessed as the 'W' registers.

On the FP size, both VFPv4 and NEONv2 now operate on a set of 32 128-bit FP registers (called 'V' registers). Differently compared to the ARMv7A architecture, the 64-bit 'D' registers are now the lower half of the 'V' registers, and the 32-bit 'S' registers are the lower half of the 'D' registers. Therefore, 32 FP registers are available, regardless of the width of the instructions. This is meant to simplify the implementation of processors, but removes the possibility to operate on all single elements of the wide 128-bit registers. NEON gets fully IEEE compilant, and adds support to double-precision FP numbers.

5.4.1 Cortex A57

The Cortex A57 is essentially the Cortex A15 with the ARMv8A ISA. Most of the architectural features are unchanged: the core is out-of-order, superscalar (that can decode 3 instructions per clock cycle) with speculative execution and register renaming. The execution ports and the pipelines are mostly unchanged too. The most relevant change is that the L1 instruction cache gets 48 KB and 3-way associative.

The processor is still able to perform a 128-bit wide FMA every clock cycle, with a latency of 10 clock cycles. However, NEON now supports also double-precision FP numbers, meaning that 4 single-precision or 2 double-precision FMA can be performed every clock cycle. Therefore, in double precision the full FP throughput is doubled with respect to the Cortex A15, while in single precision it is unchanged.

Even if the processor is out-of-order, the code is written in assembly. This is due to the fact that the compiler `gcc 4.9` still does not produce high enough quality code (even if the situation should improve with subsequent versions).

5.4.1.1 dgemm

As an improvement over Cortex A15, the vector NEON instructions can be used also with double-precision FP numbers. Since the FMA latency is of 10 clock cycles, at least 10 V registers must be employed. Therefore, a natural `dgemm` kernel is 8×4 (with $b_s = 4$), that makes use of 16 V registers to hold a 8×4 sub-matrix of the result matrix. The remaining 16 registers are more than enough to hold the A and B elements for two consecutive iterations over k, allowing for an aggressive prefetch. Additionally, the use of prefetch instruction `prfm` (with hint for persistent data in L1) is found to slightly improve performance.

The code for one iteration over k lools like

```
 1: ld1   {v6.2d, v7.2d}, [x1], #32
 2: fmla  v16.2d, v0.2d, v4.2d[0]
 3: ld1   {v10.2d, v11.2d}, [x3], #32
 4: fmla  v17.2d, v1.2d, v4.2d[0]
 5: ld1   {v8.2d, v9.2d}, [x11], #32
 6: fmla  v24.2d, v2.2d, v4.2d[0]
 7: prfm  PLDL1KEEP, [x3, #64]
 8: fmla  v25.2d, v3.2d, v4.2d[0]

 9: prfm  PLDL1KEEP, [x1, #64]
10: fmla  v18.2d, v0.2d, v4.2d[1]
11: prfm  PLDL1KEEP, [x11, #64]
12: fmla  v19.2d, v1.2d, v4.2d[1]
13: fmla  v26.2d, v2.2d, v4.2d[1]
14: fmla  v27.2d, v3.2d, v4.2d[1]

15: fmla  v20.2d, v0.2d, v5.2d[0]
16: fmla  v21.2d, v1.2d, v5.2d[0]
17: fmla  v28.2d, v2.2d, v5.2d[0]
18: fmla  v29.2d, v3.2d, v5.2d[0]

19: fmla  v22.2d, v0.2d, v5.2d[1]
20: fmla  v23.2d, v1.2d, v5.2d[1]
21: fmla  v30.2d, v2.2d, v5.2d[1]
22: fmla  v31.2d, v3.2d, v5.2d[1]
```

where `fmla` is the FMA instruction, `ld1` is the register load instruction and `prfm` is the prefetch hint instruction. The suffix `.2d` after the vector name specifies the number and size of the vector elements (2 double-precision FP numbers per

vector).

5.4.1.2 sgemm

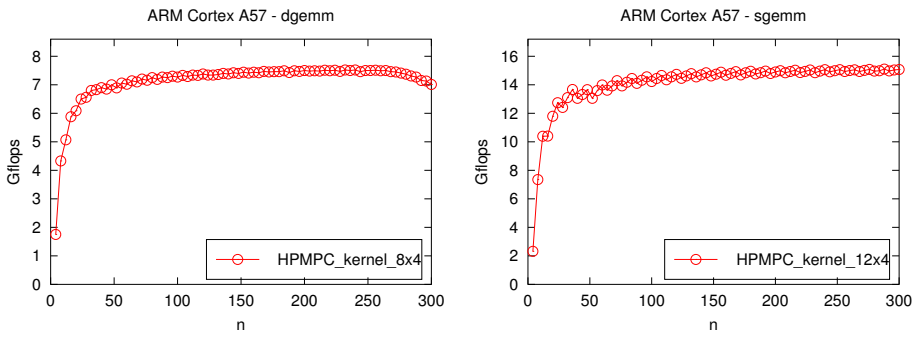
In single precision, the code is essentially the same as the Cortex A15, translated to the new ISA and exploiting the availability of more registers. Therefore, the `sgemm` kernel is 12×4 (with $b_s = 4$), meaning that 12 V registers are employed to hold a 12×4 sub-matrix of the result matrix. The remaining 20 registers are more than enough to hold the A and B elements for 4 consecutive iterations over k, allowing for an aggressive prefetch.

Also in the single-precision case the use of prefetch instruction `prfm` (with hint for persistent data in L1) is found to slightly improve performance.

The code for one iteration over k looks like

```
 1: fmla  v16.4s, v0.4s, v6.4s[0]
 2: ld1   {v8.4s, v9.4s}, [x1], #32
 3: fmla  v17.4s, v0.4s, v6.4s[1]
 4: ld1   {v14.4s, v15.4s}, [x3], #32
 5: fmla  v18.4s, v0.4s, v6.4s[2]
 6: ld1   {v10.4s, v11.4s}, [x11], #32
 7: fmla  v19.4s, v0.4s, v6.4s[3]
 8: ld1   {v12.4s, v13.4s}, [x14], #32

 9: fmla  v20.4s, v2.4s, v6.4s[0]
10: prfm  PLDL1KEEP, [x1, #64]
11: fmla  v21.4s, v2.4s, v6.4s[1]
12: prfm  PLDL1KEEP, [x3, #64]
13: fmla  v22.4s, v2.4s, v6.4s[2]
14: prfm  PLDL1KEEP, [x11, #64]
15: fmla  v23.4s, v2.4s, v6.4s[3]
16: prfm  PLDL1KEEP, [x14, #64]

17: fmla  v24.4s, v4.4s, v6.4s[0]
18: fmla  v25.4s, v4.4s, v6.4s[1]
19: fmla  v26.4s, v4.4s, v6.4s[2]
20: fmla  v27.4s, v4.4s, v6.4s[3]
```

The instruction name is the same as in double precision. The suffix `.4s` after



(a) Cortex A57 dgemm. **(b)** Cortex A57 sgemm.

Figure 5.8: Performance test of `gemm` kernel for squared matrices of size $n \times n$, $n \in [4, 300]$, on an ARM Cortex A57, code compiled with gcc 4.?. Peak performance in double (single) precision is $4 \cdot 2.15 = 8.6$ Gflops ($8 \cdot 2.15 = 17.2$ Gflops).

the vector name specifies the number and size of the vector elements (4 single-precision FP numbers per vector).

Given the large number of registers, other `sgemm` kernel configurations are possible, such as 8×8 (with both $b_s = 4$ and $b_s = 8$), however, the 12×4 configuration is preferred to make easier the reuse of code with Cortex A15.

5.4.1.3 Results

The test processor is one core of the NVIDIA Tegra X1 SoC (running at 2.15 GHz) in the Shield TV. The memory is 3 GB of LPDDR4-3200 RAM giving 25.6 GB of bandwidth. As shown in Figure 5.8, both in double and in single precision the best performance is of about 87% of full FP throughput. The figure is single precision is essentially identical to the Cortex A15 one.

At the time of writing, the latest stable release of OpenBLAS still does not support Cortex A57. The support is being added to the latest code version on GitHub, but the code is buggy and can not compile successfully.

5.5 PowerPC

PowerPC (standing for Performance Optimization With Enhanced RISC - Performance Computing) is a RISC ISA created in 1991 by the Apple-IBM-Motorola alliance. The ISA has evolved over time, and in 2006 has been renamed Power ISA. PowerPC is largely based on the previous Power ISA developed by IBM.

5.5.1 PowerPC 603e

The PowerPC 603e is part of the second generation of processors implementing the PowerPC ISA. The PowerPC 603 is the first processor implementing the complete 32-bit PowerPC architecture. It is designed for low cost and low power consumption. It features 8 KB of L1 data and instruction caches. The PowerPC 603e is an enhanced version, featuring 16 KB of L1 data and instruction caches.

The PowerPC 603e core is superscalar, and it can issue and retire up to 3 instructions per clock cycle, while executing 5 instructions on 5 execution units. It supports out-of-order execution and register renaming. There is a FPU operating on 32 64-bit registers, and implementing a fused-multiply-add instructions (FMA). The FPU is fully IEEE 754-compliant for both single- and double-precision operations. The PowerPC 603e does not have a SIMD unit.

Each FP register can hold either a single or a double precision FP. Therefore, the implementation scheme is analogue for single and double precision: out of the 32 FP registers, 16 are used as accumulation registers to hold a 4×4 submatrix of the result matrix D. The remaining registers are used to hold and to aggressively prefetch elements from A and B.

5.5.1.1 dgemm

In double precision, a FMA instruction can be issued every other clock cycle. Therefore the full FP throughput is of 1 flop per cycle.

Generic C code compiled with the `gcc` compiler seems to work rather well with the PowerPC 603e processor. However, an optimized assembly version has also been developed, since it may be useful in case of limited optimization options available on embedded devices.

The optimized code for an iteration over k is

```
 1: fmadd  0,16,20,0
 2: fmadd  1,17,20,1
 3: fmadd  2,18,20,2
 4: fmadd  3,19,20,3
 5: lfd    20,64(%3)

 6: fmadd  4,16,21,4
 7: fmadd  5,17,21,5
 8: fmadd  6,18,21,6
 9: fmadd  7,19,21,7
10: lfd    21,72(%3)

11: fmadd  8,16,22,8
12: fmadd  9,17,22,9
13: fmadd  10,18,22,10
14: fmadd  11,19,22,11
15: lfd    22,80(%3)

16: fmadd  12,16,23,12
17: lfd    16,64(%2)
18: fmadd  13,17,23,13
19: lfd    17,72(%2)
20: fmadd  14,18,23,14
21: lfd    18,80(%2)
22: fmadd  15,19,23,15
23: lfd    23,88(%3)
24: lfd    22,88(%3)
```

This code closely resembles the code for other RISC ISAs with scalar FP unit, such as the code for double precision ARMv7A processors. The main difference (a part from the instruction name) is the fact that PowerPC FMA instructions have 4 arguments.

5.5.1.2 sgemm

In single precision, a FMA instruction can be issued every clock cycle, twice as much as in double precision. Therefore the full FP throughput is of 2 flops per cycle.

Generic C code compiled with the `gcc` compiler seems to work rather well with the PowerPC 603e processor. However, an optimized assembly version has also

been developed, since it may be useful in case of limited optimization options available on embedded devices.

The optimized code for an iteration over k is

```
 1: fmadds 0,16,20,0
 2: fmadds 1,17,20,1
 3: fmadds 2,18,20,2
 4: fmadds 3,19,20,3
 5: lfs    20,32(%3)

 6: fmadds 4,16,21,4
 7: fmadds 5,17,21,5
 8: fmadds 6,18,21,6
 9: fmadds 7,19,21,7
10: lfs    21,36(%3)

11: fmadds 8,16,22,8
12: fmadds 9,17,22,9
13: fmadds 10,18,22,10
14: fmadds 11,19,22,11
15: lfs    22,40(%3)

16: fmadds 12,16,23,12
17: lfs    16,32(%2)
18: fmadds 13,17,23,13
19: lfs    17,36(%2)
20: fmadds 14,18,23,14
21: lfs    18,40(%2)
22: fmadds 15,19,23,15
23: lfs    23,44(%3)
24: lfs    22,44(%3)
```

The code is basically identical to the double precision case.

5.5.1.3 Results

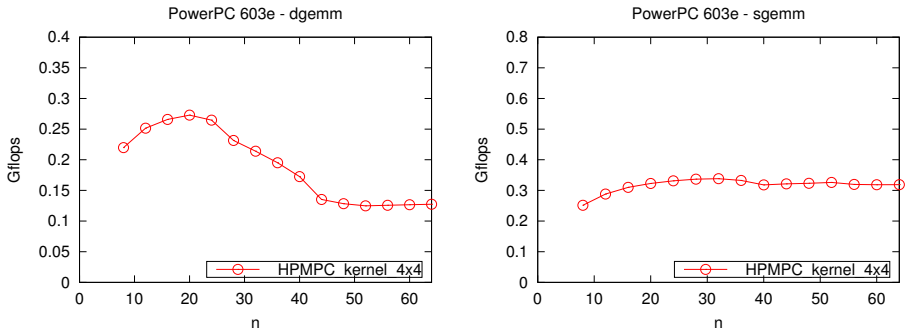
The PowerPC target platform is the ABB AC500 PM592-ETH programmable logic controller (PLC), which has a Freescale MPC8247CVRTIEA microcontroller (SoC). The core is the G2_LE implementation of the MPC603e microprocessor. The test PLC is equipped with 4MB RAM for user program memory

and 4MB integrated user data memory. In the tested PLC configuration, it is not possible to link a library, and therefore it is not possible to test the OpenBLAS library. The proposed implementation scheme for linear algebra is much simpler and therefore it is possible to embed it in the main program, without need for libraries linking. The test range is limited to matrix size n between 4 and 64 (instead of up to 300 as with the other test machines): this is due to the fact that performing test on the given PLC is much more time consuming.

The G2_LE core is a low-power (1.5W) 32-bit RISC processor running at 400 MHz. It is equipped with independent on-chip, 16 KB, 4-way set-associative, physically addressed L1 caches for instructions and data, and also on-chip instruction and data memory management units (MMUs). The cache line size is 32 byte.

In double precision (Figure 5.9a), the full FP throughput is of 0.4 Gflops. A triple-loop version can attain a good performance only for very small matrices, and performance drops significantly when the data has to be fetched from main memory (and especially for matrix size multiple of 32, due to the 4-way associativity of cache). The use of a 4×4 kernel gives a slight performance boost for matrices fitting into cache, but more importantly, it helps considerably when the memory footprint exceed cache size, since every element of A and B is used 4 times once in registers. Interestingly, the assembly coded kernel does not improve performance: FMA and the large number of registers make optimization easy, so `gcc` with `-O2` already produces good code. Nevertheless, an optimized assembly code helps in case the overall code cannot be compiled with optimization flags. There are no advantages using prefetch. The maximum performance is 0.28 Gflops (70% of full FP throughput).

In single precision (Figure 5.9b), the full FP throughput is of 0.8 Gflops. The results from the test are rather different compared to double precision. The first impression is that the performance graphs are much flatter, without the typical performance peak for data fitting in cache: the best attained performance is 0.349 Gflops (43.6% of full FP throughput). Performance tests point toward the instruction fetching as the bottleneck: in fact, for $n = 32$, leaving only FMAs in the kernel loop coded in assembly, the performance ramps up to 0.45 Gflops, but leaving only memory operations the kernel execution time halves again, so memory movement is not the bottleneck either. The G2_LE core reference manual reports that the core can sustain 2 instruction fetches per clock cycle, and a memory load and a FMA can execute in parallel every clock cycle. In practice, however, the fact that the combination of loads and FMAs is slower than each of them alone is a strong argument that the core cannot co-issue load and FMA. In this framework, the performance gain of kernels compared to triple-loop is due to the lower number of memory instructions, rather than memory movements.



(a) PowerPC 603e dgemm.

(b) PowerPC 603e sgemm.

Figure 5.9: Performance test of `gemm` kernel for squared matrices of size $n \times n$, $n \in [4, 64]$, on a PowerPC 603e code compiled with gcc 4.?. Peak performance in double (single) precision is $1 \cdot 0.4 = 0.4$ Gflops ($2 \cdot 0.4 = 0.8$ Gflops).

5.6 Conclusion

This chapter contains a collection of `gemm` kernels optimized for several widespread micro-architectures, in both single and double precision. Many relevant details about the micro-architectures are given, and the implementation choices are justified using architectural details whenever possible. Table 5.1 summarizes key features of the considered micro-architectures as well as `gemm` implementation parameters such as optimal kernel size and panel height in the panel-major matrix format.

The performance of the `gemm` kernel on the test machines spreads over more than two orders of magnitude, between roughly 0.3 Gflops for the `dgemm` kernel on the PowerPC PLC up to almost 100 Gflops for the `sgemm` kernel on the Haswell laptop. The performance of the `gemm` kernel is a good indicator (and an upper limit) to the performance of all level 3 BLAS and LAPACK routines, and of optimization algorithms implemented on top of them.

Table 5.1: Table of micro-architecture features related to the `gemm` implementation. (*) if no FMA instr. available, the throughput is the higher of multiplication instruction and addition instruction throughputs. ($) if no FMA instr. available, the latency is the sum of the multiplication instruction and the addition instruction latencies.

architecture	precision	ISA	SIMD width	(*) FMA throughput	($) FMA latency	flops /cycle	FP registers	kernel size	b_s
Intel Bonnell	single	SSE	4	2	5+5	4	8	4×4	4
(x86)	double	SSE2	1	2	5+5	1	8	2×2	2
Intel Core	single	SSE	4	1	4+3	8	16	8×4	4
(x86_64)	double	SSE3	2	1	5+3	4	16	4×4	4
Intel Nehalem	single	SSE	4	1	4+3	8	16	8×4	4
(x86_64)	double	SSE3	2	1	5+3	4	16	4×4	4
Intel Sandy-Bridge	single	AVX	8	1	5+3	16	16	16×4	8
(x86_64)	double	AVX	4	1	5+3	8	16	8×4	4
Intel Haswell	single	AVX2-FMA3	8	1/2	5	32	16	24×4	8
(x86_64)	double	AVX2-FMA3	4	1/2	5	16	16	12×4	4
Intel Skylake	single	AVX2-FMA3	8	1/2	4	32	16	24×4	8
(x86_64)	double	AVX2-FMA3	4	1/2	4	16	16	12×4	4
AMD K10	single	SSE	4	1	4+4	8	16	8×4	4
(x86_64)	double	SSE3	2	1	4+4	4	16	4×4	4
ARM Cortex A9	single	NEON	4	2	11(10)	4	16	8×4	4
(ARMv7A)	double	VFPv3	1	2	9	1	32	4×4	4
ARM Cortex A15	single	NEONv2	4	1	10?	8	16	12×4	4
(ARMv7A)	double	VFPv4	1	1	9?	2	32	4×4	4
ARM Cortex A7	single	NEONv2	4	4	?	2	16	8×4	4
(ARMv7A)	double	VFPv4	1	4	?	0.5	32	4×4	4
ARM Cortex A57	single	NEONv2	4	1	10	8	32	12×4	4
(ARMv8A)	double	NEONv2	2	1	10	4	32	8×4	4
PowerPC 603e	single	PowerPC	1	1	?	2	32	4×4	4
(PowerPC)	double	PowerPC	1	2	?	1	32	4×4	4

Chapter 6

Summary and considerations about code generation

The first part of the thesis presents dense linear algebra implementation techniques specially tailored to embedded optimization (and therefore for small-medium size matrices).

The innermost loop in each linear algebra routine is implemented as a separate kernel, hand-optimized for the specific architecture (ISA and micro-architecture). In the implementation of the kernel, blocking for registers is employed to hide the latency of FP instructions and to reuse matrix elements once in registers (decreasing the memory bandwidth requirements). Blocking for the different cache levels is not employed, given the focus on small-scale performance. The kernel code explicitly targets the specific machine instructions by means of inline assembly (for in-order architectures) or intrinsics (for out-of-order architectures); both of them give a rather fine control over the instruction choice, and the former also gives a fine control over the instruction scheduling and registers allocation, while the latter leaves these aspects to the compiler (as they are non critical in out-of-order architectures). In the case of the `gemm` routine, this kernel closely resembles the inner kernel proposed by BLIS [86] as a simplification of GotoBLAS's kernel [44].

The key difference with respect to optimized BLAS libraries is the employment of a special matrix format (named panel-major matrix format) that resembles the innermost packing structure of the data buffer employed in optimized BLAS routines. Namely, the data is stored in the same order as it is streamed by the `gemm` kernel. This matrix format is assumed as the default matrix format for all operands, and therefore no conversions of matrices between standard column-major or row-major orders needs to be done on-line. This greatly improves the performance for small matrices. However, this approach also has a few drawbacks. Since the panel-major matrix format is assumed as the default matrix format, all matrices need to have the same panel height b_s. This poses limitations on the choice of the optimal `gemm` kernel size: the number of rows and columns of the result sub-matrix processed by the kernel must be an integer multiple of the panel height. If the processed matrix is not squared, as if it has size 8×4 with panel height 4, it means that two panels from A and one from B are streamed to compute $C \leftarrow C + A \cdot B^T$, while in optimized BLAS libraries this would be optimally handled by employing 8 has panel height for A and 4 as panel height for B. Furthermore, there are difficulties when operating on sub-matrices, that in the current implementation in HPMPC can be done to a very small extent.

A specialized kernel is employed in the implementation of all required BLAS and LAPACK routines. This differs form the implementation of the LAPACK library, where small matrices are handled by unblocked routines (implemented employing level 2 BLAS, as e.g. for the `potf2` routine for the Cholesky factorization) and large matrices are handled by blocked routines (implemented employing level 3 BLAS and unblocked LAPACK routines, as e.g. for the `potrf` routine for the Cholesky factorization). In the proposed approach, there is no distinction between blocked and unblocked routines, and only a single routine is implemented for each LAPACK operation, employing specialized kernels having the `gemm` kernel as their main loop. This means that LAPACK routines for e.g. Cholesky factorization and triangular matrix inversion are implemented as if they were level 3 BLAS routines. The main advantage of the proposed approach is that it gives high performance also for small matrices, whereas standard LAPACK routines give good performance only for much larger matrices, since they employ level 3 BLAS for the operations on sub-matrices.

When dealing with embedded devices, an important aspect is the code memory size. This has not been discussed yet in this thesis: the reason for this is that the code size in HPMPC is fixed, since it is not code-generated. Furthermore, if needed it is possible to trade-off code size with performance, e.g. employing a smaller number of tailored kernels at the expense of a small reduction in computational performance. However, this is not an automated process, and, if needed, it requires some hand tuning.

Code generation has not been considered, being orthogonal to the implementation techniques presented in this thesis. However, if considered beneficial it could be easily added, as briefly discussed in the following tests.

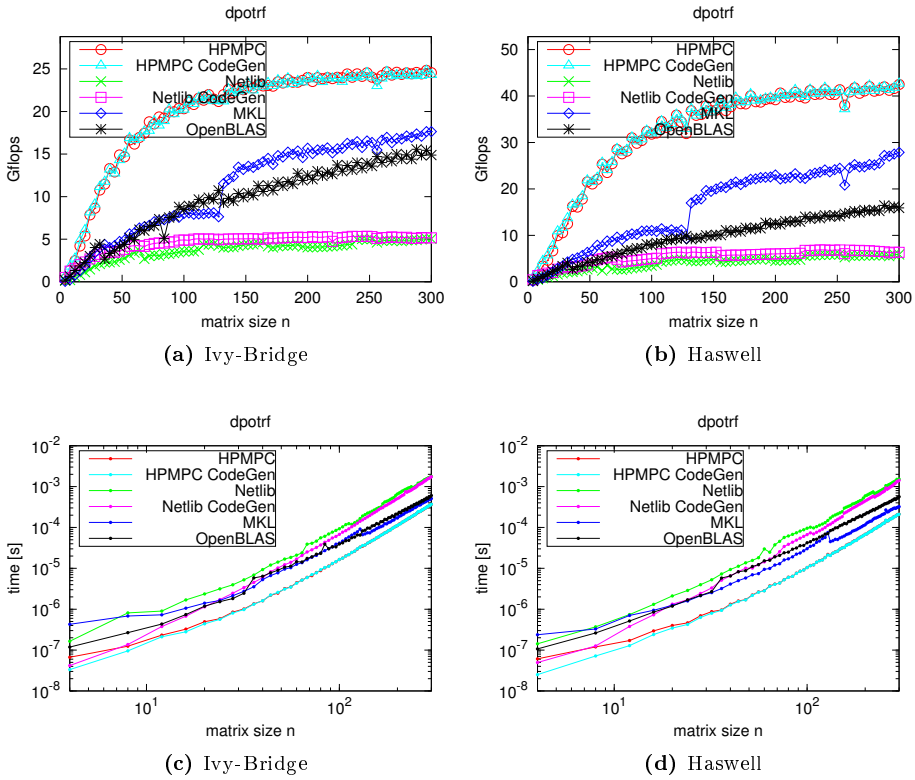
6.1 Comparison with existing dense linear algebra implementations

This section adds HPMPC (the library containing the linear algebra and optimization routines implemented using the proposed techniques) to the comparisons made in Section 2.4. The test problem is still the computation of the lower triangular Cholesky factor in double precision (given by the `dpotrf` LAPACK routine).

More precisely, two additional test are added, one for the Cholesky factorization routine as implemented in the HPMPC library (i.e., in library form, working for matrices of any size), and one for a code-generated version of the routine in HPMPC. This code-generated version employs C preprocessing to remove unnecessary branches and fix the size of the two outer loops. The linear algebra kernels have not been changed, thus keeping them suitable for code reuse. Beside the flags to enable architecture-specific instructions, the resulting code is compiled with the flags `-O3 -funroll-loops` (found to be beneficial in case of code-generated code), whereas the library version in HPMPC is compiled with the flag `-O2`. Therefore, this routine employs code-generation a la FORCES, in addition to all implementation techniques proposed in this thesis and employed in HPMPC.

Figure 6.1 contains the final comparisons of this first part of the thesis. The main result is that the Cholesky factorization routine in HPMPC gives consistently better performance than all alternatives, on both the tested Ivy-Bridge and Haswell architectures. For very small matrices, the code-generated triple loop version gives slightly better performance that the library version in HPMPC, with break-even point around 8. However, the code-generated version of the routine in HPMPC gives the absolute better performance. For matrices of size in the range 10-100, code generation does not add any noticeable performance to the routine in HPMPC, that clearly stands out being 2-6 times faster than all other routines. For larger matrices, the performance of optimized BLAS libraries approaches the performance of the routine in HPMPC (and eventually exceed it for even large matrices), while triple-loop based routines (irrespectively of code generation) perform clearly worse.

As a conclusion, the implementation techniques presented in the first part of



(a) Ivy-Bridge **(b)** Haswell

(c) Ivy-Bridge **(d)** Haswell

Figure 6.1: Performance test for the LAPACK `dpotrf` routine on an Intel Core i7 3520M processor (Ivy Bridge micro-architecture, supporting the AVX ISA) and an Intel Core i7 4800MQ (Haswell micro-architecture, supporting the AVX2 and FMA ISAs).

this thesis can be employed to implement dense linear algebra routines giving excellent performance in the range of matrix sizes of interest in embedded optimization. Their performance exceeds both the performance of optimized BLAS libraries (especially for very small matrices) and the performance of the code-generated linear algebra routines currently employed in embedded optimization (especially for matrices with size larger than a few tens).

Code generation could be combined with the implementation techniques presented in the first part of this thesis, but the performance improvements are much smaller than in the case of triple-loop based linear algebra and generally not worthwhile. Therefore, code generation is not necessary to achieve high-performance for matrix sizes of interest in embedded optimization.

Part II

Algorithms for Unconstrained MPC and MHE Problems

Chapter 7

Unconstrained MPC and MHE problem formulations

Part II deals with efficient solution methods for the unconstrained (linear) MPC and MHE problems. These problems play a key role in optimal control. In fact, many sub-problems in optimization algorithms for constrained and non-linear MPC and MHE problems can be written as instances of unconstrained MPC and MHE problems. Therefore, the availability of fast routines for these unconstrained MPC and MHE problems allows the development of a whole class of efficient solvers for more complex optimization problems.

All solvers presented in later chapters in this part are based on the unconstrained MPC and MHE problem formulations introduced in the remainder of the section.

7.1 Unconstrained MPC problem

In this thesis, the following formulation of the unconstrained MPC problem (also known in literature as extended linear-quadratic control problem [51, 52])

is considered

$$\min_{u,x} \quad \sum_{n=0}^{N-1} \frac{1}{2} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix}^T \begin{bmatrix} R_n & S_n & r_n \\ S_n^T & Q_n & q_n \\ r_n^T & q_n^T & 0 \end{bmatrix} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} x_N \\ 1 \end{bmatrix}^T \begin{bmatrix} Q_N & q_N \\ q_N^T & 0 \end{bmatrix} \begin{bmatrix} x_N \\ 1 \end{bmatrix} \tag{7.1a}$$

$$s.t. \quad x_{n+1} = A_n x_n + B_n u_n + b_n, \quad n = 0, \dots, N-1 \tag{7.1b}$$

$$x_0 = \hat{x}_0 \tag{7.1c}$$

$$0 = D_N x_N + d_N \tag{7.1d}$$

This is an instance of equality-constrained quadratic program with a special structure. The cost function (7.1a) contains N quadratic and linear terms in the states and controls (stages 0 to $N-1$), plus a quadratic and linear terminal cost on states at stage N. The system dynamic is described by the linear state-space system (7.1b). In the control problem, the state at stage 0 is fixed (7.1c).

Sufficient conditions for the existence and uniqueness of the solution of (7.1 are that the matrices $\begin{bmatrix} R_n & S_n \\ S_n^T & Q_n \end{bmatrix}$ and Q_N are positive semi-definite, and the matrices R_n are positive definite [31].

This formulation differs from the usual unconstrained MPC formulation thanks to the additional state equality constraint on the last stage. This term can be useful to enforce stability of the MPC formulation [63]. Some solution methods can deal with this terminal constraint at a very small computational cost increase.

The size of problem (7.1) is defined by the quantities: n_x (state vector size), n_u (input vector size), n_d (number of equality constraints on the last stage), N (horizon length).

7.1.1 Marix formulation

The unconstrained MPC problem can be reformulated in the matrix form

$$\min_{\bar{y}} \quad \frac{1}{2} \bar{y}^T \bar{\mathcal{H}} \bar{y} + \bar{g}^T \bar{y} + \gamma \tag{7.2a}$$

$$s.t. \quad \bar{\mathcal{A}} \bar{y} = \bar{b}_d \tag{7.2b}$$

where (choosing $N = 3$ to simplify the notation)

$$\bar{y} = \left[\begin{array}{c} u_0 \\ u_1 \\ u_2 \\ \hline x_1 \\ x_2 \\ x_3 \end{array}\right] = \left[\begin{array}{c} \bar{u} \\ \hline \bar{x} \end{array}\right], \quad \bar{\mathcal{H}} = \left[\begin{array}{ccc|ccc} R_0 & & & S_1 & & \\ & R_1 & & & S_2 & \\ & & R_2 & & & \\ \hline & S_1^T & & Q_1 & & \\ & & S_2^T & & Q_2 & \\ & & & & & Q_3 \end{array}\right] = \left[\begin{array}{c|c} \bar{R} & \bar{S} \\ \hline \bar{S}^T & \bar{Q} \end{array}\right],$$

$$\bar{g} = \left[\begin{array}{c} S_0\hat{x}_0 + r_0 \\ r_1 \\ r_2 \\ \hline q_1 \\ q_2 \\ q_3 \end{array}\right] = \left[\begin{array}{c} \bar{r} \\ \hline \bar{q} \end{array}\right], \quad \bar{b}_d = \left[\begin{array}{c} A_0\hat{x}_0 + b_0 \\ b_1 \\ b_2 \\ \hline d_3 \end{array}\right] = \left[\begin{array}{c} \bar{b} \\ \hline \bar{d} \end{array}\right], \tag{7.3}$$

$$\bar{\mathcal{A}} = \left[\begin{array}{ccc|ccc} -B_0 & & & I & & \\ & -B_1 & & -A_1 & I & \\ & & -B_2 & & -A_2 & I \\ \hline & & & & & -D_3 \end{array}\right] = \left[\begin{array}{c|c} -\bar{B} & \bar{A} \\ \hline & \bar{D} \end{array}\right].$$

Notice that in the MPC case the matrix $\bar{A}$ is clearly invertible, since it is lower triangular with 1 on all diagonal elements.

7.1.2 Optimality conditions

The KKT optimality conditions for the unconstrained MPC problem can be found e.g. in [31].

The KKT system associated to the unconstrained MPC problem can be written in the band-diagonal form (for $N = 3$)

$$\left[\begin{array}{cc|ccc|ccc|cc} R_0 & B_0^T & & & & & & & u_0 & -S_0\hat{x}_0 - r_0 \\ B_0 & & -I & & & & & & \lambda_0 & -A_0\hat{x}_0 - b_0 \\ \hline & -I & Q_1 & S_1^T & A_1^T & & & & x_1 & -q_1 \\ & & S_1 & R_1 & B_1^T & & & & u_1 & -r_1 \\ & & A_1 & B_1 & & -I & & & \lambda_1 & -b_1 \\ \hline & & & & -I & Q_2 & S_2^T & A_2^T & x_2 & -q_2 \\ & & & & & S_2 & R_2 & B_2^T & u_2 & -r_2 \\ & & & & & A_2 & B_2 & & \lambda_2 & -b_2 \\ \hline & & & & & & & -I & x_3 & -q_3 \\ & & & & & & & D_3 & \lambda_3 & -d_3 \end{array}\right] \tag{7.4}$$

7.2 Unconstrained MHE problem

The aim of the MHE problem is the reconstruction of the state vectors x_n, process noise vectors w_n and measurement noise vectors v_n, given the plant model, the measurement vectors y_n for a window of past time instants $n = 0, 1, \ldots, N$ and an initial estimate of the state vector at time 0, $\bar{x}_0$, and relative covariance matrix $\widetilde{P}_0$, summarizing the contribution given by the measurements prior to time 0.

The unconstrained MHE problem is traditionally written as the equality-constrained quadratic program

$$\begin{aligned}\min_{x_n,w_n,v_n} \quad & \sum_{n=0}^{N} \frac{1}{2}(v_n - \bar{v}_n)^T \widetilde{R}_n^{-1}(v_n - \bar{v}_n) + \sum_{n=0}^{N-1} \frac{1}{2}(w_n - \bar{w}_n)^T \widetilde{Q}_n^{-1}(w_n - \bar{w}_n) + \\ & + \frac{1}{2}(x_0 - \bar{x}_0)^T \widetilde{P}_0^{-1}(x_0 - \bar{x}_0) \\ s.t. \quad & x_{n+1} = A_n x_n + G_n w_n + f_n, \quad n = 0, \ldots, N-1 \\ & y_n = C_n x_n + v_n \end{aligned} \tag{7.5}$$

In this formulation, the inverse of the matrices in the cost function have a precise statistical interpretation: $\widetilde{R}_n$ is the covariance matrix of the measurement noise vector v_n, $\widetilde{Q}_n$ is the covariance matrix of the process noise vector w_n. The vectors $\bar{v}_n$ and $\bar{w}_n$ are the expected values of the measurement and process noises.

In this thesis, the following more general formulation of the MHE problem is considered

$$\min_{x_n,u_n} \quad \sum_{n=0}^{N-1} \frac{1}{2} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix}^T \begin{bmatrix} R_n & S_n & r_n \\ S_n^T & Q_n & q_n \\ r_n^T & q_n^T & 0 \end{bmatrix} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} x_N \\ 1 \end{bmatrix}^T \begin{bmatrix} Q_N & q_N \\ q_N^T & 0 \end{bmatrix} \begin{bmatrix} x_N \\ 1 \end{bmatrix} \tag{7.6a}$$

$$s.t. \quad x_{n+1} = A_n x_n + B_n u_n + b_n, \quad n = 0, \ldots, N-1 \tag{7.6b}$$

$$0 = D_N x_N + d_N \tag{7.6c}$$

The MHE problem (7.5) can be rewritten in this formulation by setting

$$\begin{gathered} u_n = w_n, \quad B_n = G_n, \quad b_n = f_n, \quad D_N = 0, \quad d_N = 0, \\ Q_0 = C_0^T \widetilde{R}_0^{-1} C_0 + \widetilde{P}_0^{-1}, \quad Q_n = C_n^T \widetilde{R}_n^{-1} C_n, \quad S_n = 0, \quad R_n = \widetilde{Q}_n^{-1}, \\ q_0 = C_0^T(\bar{v}_0 - y_0) - \widetilde{P}_0^{-1}\bar{x}_0, \quad q_n = C_n^T(\bar{v}_n - y_n), \quad r_n = -\bar{w}_n. \end{gathered}$$

Formulation (7.6) reflects the deterministic view of the MHE as the problem of finding the optimal x_n, w_n and v_n sequences in a least-square sense, with respect to some cost function. This formulation looks like the MPC formulation (7.1), with one key difference: in the MHE problem the initial state x_0 is retained as an optimization variable, while in the MPC problem it is fixed to some initial value $\hat{x}_0$.

It is possible to consider additional state equality constraint on the last stage (7.6c) and still maintain a problem structure that can be easily exploited in some tailored solver at a very small computational cost increase. These equality constraints are used to provide consistent feedback signal to the controller. They are enforced only at the last stage to avoid Linear Independence Constraint Qualification (LICQ) problems.

The penalization of x_n in place of v_n in the cost function (7.6a) is useful to account for QPs in non-linear MHE. The fact that the matrices in the cost function appear as not-inverted makes straightforward the use of a solver for this MHE formulation as a routine for constrained MHE (e.g., in an IPM these matrices are updated to take into account constraints). In fact, the inversion does not need to be performed explicitly, as it can be embedded in the solution algorithms and therefore performed implicitly.

The size of problem (7.6) is defined by the quantities: n_x (state vector size), n_u (process noise vector size), n_d (number of state equality constraints on the last stage), N (horizon length).

7.2.1 Matrix formulation

The unconstrained MHE problem can be reformulated in the matrix form

$$\min_{\bar{y}} \quad \frac{1}{2}\bar{y}^T\bar{\mathcal{H}}\bar{y} + \bar{g}^T\bar{y} + \gamma \tag{7.7a}$$

$$s.t. \quad \bar{\mathcal{A}}\bar{y} = \bar{b}_d \tag{7.7b}$$

where (choosing $N = 3$ to simplify the notation)

$$\bar{y} = \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ \hline x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} \bar{u} \\ \hline \bar{x} \end{bmatrix}, \quad \bar{\mathcal{H}} = \left[\begin{array}{ccc|ccccc} R_0 & & & S_0 & & & & \\ & R_1 & & & S_1 & & & \\ & & R_2 & & & S_2 & & \\ \hline S_0^T & & & Q_0 & & & & \\ & S_1^T & & & Q_1 & & & \\ & & S_2^T & & & Q_2 & & \\ & & & & & & Q_3 \end{array}\right] = \left[\begin{array}{c|c} \bar{R} & \bar{S} \\ \hline \bar{S}^T & \bar{Q} \end{array}\right],$$

$$\bar{g} = \begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ \hline q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} = \begin{bmatrix} \bar{r} \\ \hline \bar{q} \end{bmatrix}, \quad \bar{b}_d = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \hline d_3 \end{bmatrix} = \begin{bmatrix} \bar{b} \\ \hline d \end{bmatrix},$$

$$\bar{\mathcal{A}} = \left[\begin{array}{ccc|ccccc} -B_0 & & & -A_0 & I & & & \\ & -B_1 & & & -A_1 & I & & \\ & & -B_2 & & & -A_2 & I & \\ \hline & & & & & & -D_3 \end{array}\right] = \left[\begin{array}{c|c} -\bar{B} & \bar{A} \\ \hline & \bar{D} \end{array}\right]. \tag{7.8}$$

Notice that in the MHE case the matrix $\bar{A}$ is clearly not invertible, since the first block-row is zero.

7.2.2 Optimality conditions

The KKT system associated to the unconstrained MHE problem can be written in the band-diagonal form (for $N = 3$)

$$\left[\begin{array}{ccc|ccc|ccc|cc} Q_0 & S_0^T & A_0^T & & & & & & & & \\ S_0 & R_0 & B_0^T & & & & & & & & \\ A_0 & B_0 & & -I & & & & & & & \\ \hline & & -I & Q_1 & S_1^T & A_1^T & & & & & \\ & & & S_1 & R_1 & B_1^T & & & & & \\ & & & A_1 & B_1 & & -I & & & & \\ \hline & & & & & -I & Q_2 & S_2^T & A_2^T & & \\ & & & & & & S_2 & R_2 & B_2^T & & \\ & & & & & & A_2 & B_2 & & -I & \\ \hline & & & & & & & & -I & Q_3 & D_3^T \\ & & & & & & & & & D_3 & \end{array}\right] \left[\begin{array}{c} x_0 \\ u_0 \\ \lambda_0 \\ \hline x_1 \\ u_1 \\ \lambda_1 \\ \hline x_2 \\ u_2 \\ \lambda_2 \\ \hline x_3 \\ \lambda_3 \end{array}\right] = \left[\begin{array}{c} -q_0 \\ -r_0 \\ -b_0 \\ \hline -q_1 \\ -r_1 \\ -b_1 \\ \hline -q_2 \\ -r_2 \\ -b_2 \\ \hline -q_3 \\ -d_3 \end{array}\right] \tag{7.9}$$

Chapter 8

Structure-exploiting recursive factorizations of the KKT matrix

In this chapter, two recursive methods for the solution of the MPC and MHE problems are presented. Namely, a backward and a forward recursive factorizations of the KKT matrix are considered, together with the appropriate backward and forward solutions. Both algorithms use a recursion to efficiently factorize the KKT matrix stage-wise, and have a complexity linear in the horizon length N.

The factorizations are derived for a generic linear optimal control problem (OCP), where the dynamical system is described by a state-space model, and the number of states and inputs can vary stage-wise. This allows to handle both MPC and MHE using the same algorithms, reducing development and implementation efforts. Nevertheless, each factorization is better suited to a specific OCP.

The backward Riccati recursion can naturally handle the MPC problem, and it has been widely used in literature [70, 66]. This recursion begins the factorization at the last stage. It distinguishes between state and input variables, but it can not exploit a diagonal Hessian of the cost function to reduce the com-

putational cost. This factorization can not directly handle additional equality constraints at the last stage: therefore, if $n_d > 0$, the backward Riccati recursion has to be embedded into an Interior Point (IPM) or Active Set (AS) method, or a Schur-complement approach has to be employed [70]. In the special case of LTI problem, if the recursion matrix is initialized with the solution of the discrete Riccati algebraic equation (DARE), then the factorized KKT matrix has a constant structure stage-wise, and therefore it can be computed in time constant in N (see Chapter 11).

The forward Schur-complement recursion can naturally handle the MHE problem. The recursion begins the factorization at the first stage. This recursion does not distinguish between state and input variables (and therefore can be seen as more general), but it can exploit a diagonal Hessian of the cost function to reduce the computational cost (that in general is higher than in the backward Riccati recursion case). This factorization has the advantage of directly handling additional equality constraints on the last stage, that can be useful to enforce stability in MPC formulations [63], or to enforce consistent state estimation in the MHE problem. In the MHE problem, this recursion has important relations with the Kalman filter, and in particular may be used to compute the information matrix of estimate. In the MPC problem, this recursion requires regularization at the first stage (as shown in Section 8.2.1.1), and therefore gives a solution with lower accuracy.

8.1 Backward riccati recursion

The backward Riccati recursion is a well-known method for the solution of the unconstrained MPC problem [70].

In its classical formulation, it reads

$$\begin{aligned} P_n = Q_n + A_n^T P_{n+1} A_n - \\ - (S_n^T + A_n^T P_{n+1} B_n)(R_n + B_n^T P_{n+1} B_n)^{-1}(S_n + B_n^T P_{n+1} A_n). \end{aligned} \quad (8.1)$$

This recursion is initialized at the last stage with $P_N = Q_N$. It can be interpreted as a stage-wise recursive factorization of the KKT matrix [70]. If the recursion is implemented using standard BLAS and LAPACK routines, the computational cost to factorize the KKT matrix is of about

$$N(4n_x^3 + 6n_x^2 n_u + 3n_x n_u^2 + \tfrac{1}{3}n_u^3) \quad (8.2)$$

flops [31], that is linear in N and cubic in the number of stage variables (n_x+n_u).

There exist a similar recursion for the linear terms in the cost function

$$\begin{aligned} p_n &= q_n + A_n^T(P_{n+1}b_n + p_{n+1}) - \\ &- (S_n^T + A_n^T P_{n+1}B_n)(R_n + B_n^T P_{n+1}B_n)^{-1}(s_n + B_n^T(P_{n+1}b_n + p_{n+1})) \end{aligned} \quad (8.3)$$

that can be interpreted as a stage-wise backward solution of the KKT system.

The solution procedure is completed by the forward recursion consisting on the computation of the optimal input u_n as time-varying affine state feedback

$$u_n = K_n x_n + k_n \quad (8.4)$$

where

$$\begin{aligned} K_n &= -(R_n + B_n^T P_{n+1}B_n)^{-1}(S_n + B_n^T P_{n+1}A_n) \\ k_n &= -(R_n + B_n^T P_{n+1}B_n)^{-1}(s_n + B_n^T(P_{n+1}b_n + p_{n+1})), \end{aligned}$$

and computation of the next state x_{n+1} by means of simulation

$$x_{n+1} = A_n x_b + B_n u_n + b_n.$$

This forward recursion can be interpreted as a stage-wise forward substitution of the KKT system.

The aim of the remainder of the section is the derivation of an efficient formulation for the backward Riccati recursion (that has a lower computational complexity with respect to the classical formulation in (8.1)), and the presentation of implementations using either the standard BLAS interface or the linear algebra routines in HPMPC.

8.1.1 Derivation

In this section, the backward Riccati recursion as a solution strategy for a linear OCP is derived using two methods: recursive factorization of the KKT matrix and dynamic programming. These two methods give different interpretations to the Riccati recursion. The interpretation as factorization of the KKT matrix justifies the use of techniques such as mixed precision computation [23] in the context of MPC [33]. The derivation using dynamic programming gives a formulation that naturally leads to an efficient implementation.

8.1.1.1 Backward Riccati recursion as factorization of the KKT matrix

The backward Riccati recursion can be seen as a block-wise recursive factorization of the KKT matrix [70]. Here the factorization procedure is repeated for pedagogical purposes.

The recursive factorization starts at the last stage. The recursion comprises a recursive step (operating on the general stages) and a end-of-recursion step (operating on the first stage).

Recursive step The backward Riccati recursion can not naturally handle equality constraints on the last stage. Therefore, it is assumed that $n_d = 0$ and that the last two stages are in the form

$$\left[\begin{array}{c|ccc|c} -I & Q_n & S_n^T & A_n^T & \\ & S_n & R_n & B_n^T & \\ & A_n & B_n & & -I \\ \hline & & & -I & P_{n+1} \end{array}\right] \left[\begin{array}{c} \lambda_{n-1} \\ \hline x_n \\ u_n \\ \lambda_n \\ \hline x_{n+1} \end{array}\right] = - \left[\begin{array}{c} q_n \\ r_n \\ b_n \\ \hline p_{n+1} \end{array}\right]. \tag{8.5}$$

Notice that the identity matrix on the top-left corner links the factorization of this block to the previous ones. The variable x_{n+1} and the fourth equation can be eliminated by adding the fourth equation to the third equation multiplied by P_{n+1}

$$\left[\begin{array}{c|ccc} -I & Q_n & S_n^T & A_n^T \\ & S_n & R_n & B_n^T \\ & P_{n+1}A_n & P_{n+1}B_n & -I \end{array}\right] \left[\begin{array}{c} \lambda_{n-1} \\ \hline x_n \\ u_n \\ \lambda_n \end{array}\right] = - \left[\begin{array}{c} q_n \\ r_n \\ P_{n+1}b_n + p_{n+1} \end{array}\right].$$

The variable λ_n and the third equation can be eliminated by adding the third equation multiplied by A_n^T to the first equation and by adding the third equation multiplied by B_n^T to the second equation

$$\left[\begin{array}{c|cc} -I & Q_n + A_n^T P_{n+1} A_n & S_n^T + A_n^T P_{n+1} B_n \\ & S_n + B_n^T P_{n+1} A_n & R_n + B_n^T P_{n+1} B_n \end{array}\right] \left[\begin{array}{c} \lambda_{n-1} \\ \hline x_n \\ u_n \end{array}\right] = - \begin{bmatrix} q_n + A_n^T (P_{n+1} b_n + p_{n+1}) \\ r_n + B_n^T (P_{n+1} b_n + p_{n+1}) \end{bmatrix}$$

Finally, in the hypothesis that R_n is positive definite and then invertible, the variable u_n and the second equation can be eliminated by means of the Schur complement of the bottom-right block, as

$$\left[\begin{array}{c|cc} -I & P_n \end{array}\right] \left[\begin{array}{c} \lambda_{n-1} \\ \hline x_n \end{array}\right] = - \begin{bmatrix} p_n \end{bmatrix}$$

where P_n and p_n have the same expressions as for the backward Riccati recursion (8.1) and (8.3). This closes the recursion, since this equation is identical to the last equation in (8.5).

End of recursion At the end of the recursion, the MPC and the MHE problems are handled in different ways.

In the MPC case, the first stage is in the form

$$\left[\begin{array}{cc|c} R_0 & B_0^T & \\ B_0 & & -I \\ \hline & -I & P_1 \end{array}\right] \left[\begin{array}{c} u_0 \\ \lambda_0 \\ \hline x_1 \end{array}\right] = - \left[\begin{array}{c} r_0 \\ b_0 \\ \hline p_1 \end{array}\right]$$

Using arguments analogue to the recursive step, this can be rewritten in the form

$$\left[R_0 + B_0^T P_1 B_0\right] \left[u_0\right] = - \left[r_0 + B_0^T (P_1 b_0 + p_1)\right] \tag{8.6}$$

In the hypothesis that R_0 is positive definite and then invertible, the matrix in (8.6) can be factorized using e.g. Cholesky factorization, and therefore completing the KKT matrix factorization.

In the MHE case, the first stage is in the form

$$\left[\begin{array}{ccc|c} Q_0 & S_0^T & A_0^T & \\ S_0 & R_0 & B_0^T & \\ A_0 & B_0 & & -I \\ \hline & & -I & P_1 \end{array}\right] \left[\begin{array}{c} x_0 \\ u_0 \\ \lambda_0 \\ \hline x_1 \end{array}\right] = - \left[\begin{array}{c} r_0 \\ b_0 \\ \hline p_1 \end{array}\right]$$

Using arguments analogue to the recursive step, this can be rewritten in the form

$$\begin{bmatrix} Q_0 + A_0^T P_1 A_0 & S_0^T + A_0^T P_1 B_0 \\ S_0 + B_0^T P_1 A_0 & R_0 + B_0^T P_1 B_0 \end{bmatrix} \begin{bmatrix} x_0 \\ u_0 \end{bmatrix} = - \begin{bmatrix} q_0 + A_0^T (P_1 b_0 + p_1) \\ r_0 + B_0^T (P_1 b_0 + p_1) \end{bmatrix} \tag{8.7}$$

In the hypothesis that the matrix $\begin{bmatrix} Q_0 & S_0^T \\ S_0 & R_0 \end{bmatrix}$ is positive definite and then invertible, the matrix in (8.7) can be factorized using e.g. Cholesky factorization, and therefore completing the KKT matrix factorization.

8.1.1.2 Backward Riccati as dynamic programming algorithm

In this section, the backward Riccati recursion is derived using a different approach: dynamic programming. This derivation naturally gives an efficient formulation of the backward Riccati recursion. This formulation is characterized

by lower asymptotic complexity and better computational performance than the classical backward Riccati recursion. This is obtained by exploiting symmetry and positive semi-definiteness of the recursion matrix P_{n+1}, and by propagating the Cholesky factor L_{n+1} of the recursion matrix. The propagation of the Cholesky factor of the recursion matrix justifies the name of the routine: square-root backward Riccati recursion. Furthermore, in this formulation the backward factorization and the backward solution are merged into a single recursion, further improving computational performance.

In the following, the concept of dynamic programming and its use to solve the linear MPC problem is assumed known: see [22] for the use of dynamic programming in optimal control.

Let us assume that the optimal stage cost at the generic stage $n+1$ is

$$V_{n+1}^*(x_{n+1}) = \begin{bmatrix} x_{n+1}^T & 1 \end{bmatrix} \begin{bmatrix} P_{n+1} & p_{n+1} \\ p_{n+1}^T & \pi_{n+1} \end{bmatrix} \begin{bmatrix} x_{n+1} \\ 1 \end{bmatrix}.$$

Inserting the expression of the state at stage $n+1$ as function of the state and the input at stage n,

$$x_{n+1} = \begin{bmatrix} B_n & A_n & b_n \end{bmatrix} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix}$$

the optimal stage cost can be written as function of the state and the input at stage n,

$$\begin{aligned} V_{n+1}^*(x_n, u_n) &= \mathcal{X}_n^T \mathcal{A}_n^T \mathcal{P}_{n+1} \mathcal{A}_n \mathcal{X}_n = \\ &= \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix}^T \begin{bmatrix} B_n^T & 0 \\ A_n^T & 0 \\ b_n & 1 \end{bmatrix} \begin{bmatrix} P_{n+1} & p_{n+1} \\ p_{n+1}^T & \pi_{n+1} \end{bmatrix} \begin{bmatrix} B_n & A_n & b_n \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix} \end{aligned}$$

In this expression, the state x_n is assumed given, while the input u_n can be freely chosen.

Note that the matrix in this expression can be computed efficiently by exploiting symmetry and positive definiteness of $\mathcal{P}_{n+1}$. In fact, if the matrix $\mathcal{P}_{n+1}$ is positive definite, it can be factorized using Cholesky factorization, as

$$\mathcal{P}_{n+1} = \mathcal{L}_{n+1} \mathcal{L}_{n+1}^T = \begin{bmatrix} L_{n+1,22} & \\ L_{n+1,32} & L_{n+1,33} \end{bmatrix} \begin{bmatrix} L_{n+1,22}^T & L_{n+1,32}^T \\ & L_{n+1,33}^T \end{bmatrix}$$

and then the optimal stage cost becomes

$$\begin{aligned} V_{n+1}^*(x_n, u_n) &= \mathcal{X}_n^T \mathcal{A}_n^T \mathcal{L}_{n+1} \mathcal{L}_{n+1}^T \mathcal{A}_n \mathcal{X}_n = \\ &= \mathcal{X}_n^T (\mathcal{A}_n^T \mathcal{L}_{n+1}) (\mathcal{A}_n^T \mathcal{L}_{n+1})^T \mathcal{X}_n \end{aligned}$$

that can be build efficiently by using specialized linear algebra routines to exploit the symmetry of the formulation and the fact that $\mathcal{L}_{n+1}$ is a lower triangular matrix.

The stage cost at the stage n (dropping in the last equation the index $_n$ in x, u, A, B, b, Q, S, R, q, s, ρ and the index $_{n+1}$ in P, p, π)

$$\begin{aligned}
&V_n(x_n, u_n) = \varphi_n(x_n, u_n) + V^*_{n+1}(x_n, u_n) = \\
&= \mathcal{X}_n^T(\mathcal{Q}_n + (\mathcal{A}_n^T\mathcal{L}_{n+1})(\mathcal{A}_n^T\mathcal{L}_{n+1})^T)\mathcal{X}_n = \\
&= \begin{bmatrix} u \\ x \\ 1 \end{bmatrix}^T \begin{bmatrix} R + B^TPB & S + B^TPA & s + B^T(Pb + p) \\ S^T + A^TPB & Q + A^TPA & q + A^T(Pb + p) \\ s^T + (Pb + p)^TB & q^T + (Pb + p)^TA & \rho + b^TPb + 2b^Tp + \pi \end{bmatrix} \begin{bmatrix} u \\ x \\ 1 \end{bmatrix}
\end{aligned}$$

is a function of x_n (fixed) and u_n (free), and it can be easily minimized with respect to u_n in the following way. The matrix is positive definite (since it is the sum of a positive definite matrix and a positive semi-definite matrix), and then the stage cost can be factorized by using the Cholesky factorization of the matrix,

$$\mathcal{M}_n = \mathcal{Q}_n + \mathcal{A}_n^T\mathcal{P}_{n+1}\mathcal{A}_n = \begin{bmatrix} L_{n,11} & & \\ L_{n,21} & L_{n,22} & \\ L_{n,31} & L_{n,32} & L_{n,33} \end{bmatrix} \begin{bmatrix} L_{n,11}^T & L_{n,21}^T & L_{n,31}^T \\ & L_{n,22}^T & L_{n,32}^T \\ & & L_{n,33}^T \end{bmatrix}$$

obtaining the expression for the stage cost $V_n(x_n, u_n)$

$$\begin{aligned}
V_n(x_n, u_n) = &\begin{bmatrix} L_{n,11}^Tu_n + L_{n,21}^Tx_n + L_{n,31}^T \\ L_{n,22}^Tx_n + L_{n,32}^T \\ L_{n,33}^T \end{bmatrix}^T \begin{bmatrix} L_{n,11}^Tu_n + L_{n,21}^Tx_n + L_{n,31}^T \\ L_{n,22}^Tx_n + L_{n,32}^T \\ L_{n,33}^T \end{bmatrix} = \\
&= (L_{n,11}^Tu_n + L_{n,21}^Tx_n + L_{n,31}^T)^T(L_{n,11}^Tu_n + L_{n,21}^Tx_n + L_{n,31}^T) + \\
&\quad + (L_{n,22}^Tx_n + L_{n,32}^T)^T(L_{n,22}^Tx_n + L_{n,32}^T) + L_{n,33}L_{n,33}^T.
\end{aligned}$$

Notice that u_n is present only in the first term of the sum: this term is a square, and then its minimum is zero, attained for the value of u_n

$$u_n = -(L_{n,11}^T)^{-1}(L_{n,21}^Tx_n + L_{n,31}^T). \tag{8.8}$$

The corresponding optimal value $V_n^*(x_n)$ of the stage cost is given by the remaining two terms of the sum:

$$\begin{aligned}
V_n^*(x_n) &= (L_{n,22}^Tx_n + L_{n,32}^T)^T(L_{n,22}^Tx_n + L_{n,32}^T) + L_{n,33}L_{n,33}^T = \\
&= \begin{bmatrix} x_n^T & 1 \end{bmatrix} \begin{bmatrix} P_n & p_n \\ p_n & \pi_n \end{bmatrix} \begin{bmatrix} x_n \\ 1 \end{bmatrix}
\end{aligned}$$

as in the classical formulation of the dynamic programming. Notice that the procedure gives a factorization of the matrix $\mathcal{P}_n$ that can be used at the following stage to efficiently compute $\mathcal{A}_{n-1}^T\mathcal{P}_n\mathcal{A}_{n-1}$.

The value of u_n in (8.8) can be rewritten as

$$u_n = -(R_n + B_n^T P_{n+1} B_n)^{-1}\big((S_n + B_n^T P_{n+1} A_n)x_n + r_n + B_n^T(P_{n+1} b_n + p_{n+1})\big) = \\ = K_n x_n + k_n$$

that is the expression of u_n as time varying affine state feedback given by the classical Riccati recursion in (8.4). However, the procedure to compute u_n as in (8.8) is more efficient from a computational point of view. Also notice that the recursion matrix P_n of the Riccati recursion is never computed explicitly in the above solution procedure.

8.1.2 Implementation

In this section, the implementation of the square-root backward Riccati recursion is presented. Firstly the algorithm is presented using standard BLAS and LAPACK routines, and the computational complexity as number of flops is derived for the algorithm. Then the algorithm is presented using the custom linear algebra routines proposed in Part I of the thesis. Finally, analogies of the algorithm with respect to array algorithms are discussed.

8.1.2.1 Implementation using BLAS and LAPACK

The square-root backward Riccati recursion can be implemented using the standard linear algebra routines in BLAS and LAPACK. The algorithm is summarized in Algorithm 1, where the name of the BLAS and LAPACK routines employed in the implementation appears as a comment.

The number of states n_x and the number of inputs n_u are assumed to be time-variant, and therefore in the following represented as C arrays of size $N+1$. In this framework, the difference between MPC and MHE problems is that MPC problems have $n_x[0] = 0$ and MHE problems have $n_x[0] \neq 0$. All matrices in Algorithm 1 are assumed to be of the proper size. As an example, in the case of a MPC problem, the matrices S_0 and $L_{0,21}$ have size $0 \times n_u[0]$, the matrices Q_0 and $L_{0,22}$ have size 0×0 and the vectors q_0 and $L_{0,32}$ have size 0.

The cost of this algorithm is (showing only terms cubic in the stage variable sizes, and assuming that the number of states and inputs is constant stage-wise, and equal to n_x and n_u respectively)

$$N\left(\tfrac{7}{3}n_x^3 + 4n_x^2 n_u + 2n_x n_u^2 + \tfrac{1}{3}n_u^3\right) + \tfrac{1}{3}n_x^3 \tag{8.9}$$

Algorithm 1 Square-root backward Riccati recursion - factorization and solution

1: $\begin{bmatrix} L_{N+1,22} & \\ L_{N+1,32} & L_{N+1,33} \end{bmatrix} \leftarrow \mathcal{P}^{1/2}$ ▷ potrf
2: **for** $n \leftarrow N-1, \ldots, 0$ **do**
3: $\quad \mathcal{A}_n^T \mathcal{L}_{n+1} \leftarrow \begin{bmatrix} B_n^T \\ A_n^T \\ b_n^T \end{bmatrix} \cdot L_{n+1,22} + \begin{bmatrix} 0 \\ 0 \\ L_{n+1,32} \end{bmatrix}$ ▷ trmm
4: $\quad \mathcal{M}_n \leftarrow \mathcal{Q}_n + (\mathcal{A}_n^T \mathcal{L}_{n+1}) \cdot (\mathcal{A}_n^T \mathcal{L}_{n+1})^T$ ▷ syrk
5: $\quad \begin{bmatrix} L_{n,11} & & \\ L_{n,21} & L_{n,22} & \\ L_{n,31} & L_{n,32} & L_{n,33} \end{bmatrix} \leftarrow \mathcal{M}_n^{1/2}$ ▷ potrf
6: **end for**
7: **if** $n_x[0] > 0$ **then**
8: $\quad x_0 \leftarrow -L_{0,22}^{-T} \cdot L_{0,32}^T$ ▷ trsv
9: **end if**
10: **for** $n \leftarrow 0, \ldots, N-1$ **do**
11: $\quad u_n \leftarrow -L_{n,11}^{-T} \cdot (L_{n,21}^T \cdot x_n + L_{n,31}^T)$ ▷ gemv & trsv
12: $\quad x_{n+1} \leftarrow \begin{bmatrix} B_n & A_n \end{bmatrix} \cdot \begin{bmatrix} u_n \\ x_n \end{bmatrix} + b_n$ ▷ gemv
13: $\quad \lambda_{n+1} \leftarrow L_{n+1,22} \cdot (L_{n+1,22}^T \cdot x_{n+1} + L_{n+1,32}^T)$ ▷ trmv & trmv
14: **end for**

flops in the MHE case, and $\frac{7}{3}n_x^3+3n_x^2n_u+n_xn_u^2$ flops less in the MPC case. That is lower than the classical backward Riccati recursion cost (8.2). A diagonal Hessian of the cost function could be exploited only at the last stage, in the computation of the Cholesky facto $L_{N+1,22}$ at line 1 of Algorithm 1, and at the following `trmm` call in line 3, reducing the cost of $\frac{4}{3}n_x^3 + n_x^2n_u$ flops. However, this case is unlikely to happen in practice, since the backward Riccati recursion is preferably employed in MPC problems, where often P is initialized to the solution of the Discrete Algebraic Riccati Equation (DARE), that is generally a dense matrix (unless a change in state-space representation that diagonalizes P is employed).

Since typically in control $n_x > n_u$, the most expensive part of the backward Riccati recursion algorithm (8.1) is the computation of the term $A_n^TP_{n+1}A_n$, where all matrices are squared of size $n_x \times n_x$. If neither symmetry nor positive-definiteness is exploited (as in the classical backward Riccati recursion), this term is computed using two calls to the BLAS routine `gemm`, for a total cost of $4n_x^3$.

On the other hand, if both symmetry and positive-definiteness are exploited as in the square-root version, the matrix P_{n+1} can be factorized using the Cholesky factorization routine `potrf` at cost $\frac{1}{3}n_x^3$, obtaining the lower triangular factor $L_{n+1,22}$. The term $A_n^TP_{n+1}A_n$ is then computed as $(A_n^TL_{n+1,22})(A_n^TL_{n+1,22})^T$, where the computation of $(A_n^TL_{n+1,22})$ requires n_x^3 flops using the specialized routine `trmm` (i.e. exploiting the fact that the matrix $L_{n+1,22}$ is lower triangular), while the multiplication of a matrix by the transposed of the matrix itself requires n_x^3 flops using the specialized routine `syrk` (i.e. exploiting the fact that the result matrix is symmetric, and then only the lower triangular part needs to be computed).

In Algorithm 1, this idea has been extended to the computation of all other terms in the Riccati recursion, fully exploiting symmetry and positive-definiteness to reduce the flops count.

The use of few big matrices as $\mathcal{A}_n$ of $\mathcal{Q}_n$ packing together all data matrices reduces the number of function calls to linear-algebra routines (and relative overhead), and improves performance (since optimized BLAS routines give higher performance on larger matrices), even leaving the number of flops unchanged.

Notice that the Cholesky factorization routine `potrf` provided by LAPACK requires the input matrix to be symmetric and strictly positive-definite. A sufficient condition for this to hold in Algorithm 1 is that matrices $\mathcal{Q}_n$ and $\mathcal{P}$ are symmetric and strictly positive-definite. This is a stricter requirement than in the classical Riccati recursion algorithm, where the matrices $\mathcal{Q}_n$ and $\mathcal{P}$ need to be symmetric but only positive semi-definite, and only the matrices R_n

are required to be strictly positive-definite to guarantee the invertibility of the matrices $R_n + B_n^T P_{n+1} B_n$ at all stages. The use of custom linear algebra can relax the positive-definiteness requirement also for Algorithm 1, if the backward factorization and the backward substitutions are not merged, i.e. if Algorithms 2 and 3 are employed in place of Algorithm 1.

If several linear OCP having the same KKT matrix and different RHS vector need to be solved, then it is convenient to factorize the KKT matrix only once by means of Algorithms 1 or 2 and reuse the factorized KKT matrix for all subsequent solutions, by means of Algorithm 3.

Algorithm 2 Square-root backward Riccati recursion - factorization

$L_{N+1,22} \leftarrow P_N^{1/2}$ ▷ potrf
for $n \leftarrow N-1, \dots, 0$ do
  $\mathcal{A}_n^T \mathcal{L}_{n+1} \leftarrow \begin{bmatrix} B_n^T \\ A_n^T \end{bmatrix} \cdot L_{n+1,22}$ ▷ trmm
  $\mathcal{M}_n \leftarrow \mathcal{Q}_n + (\mathcal{A}_n^T \mathcal{L}_{n+1}) \cdot (\mathcal{A}_n^T \mathcal{L}_{n+1})^T$ ▷ syrk
  $\begin{bmatrix} L_{n,11} & \\ L_{n,21} & L_{n,22} \end{bmatrix} \leftarrow \mathcal{M}_n^{1/2}$ ▷ potrf
end for

Algorithm 3 Square-root backward Riccati recursion - solution

$p_N \leftarrow p$
for $n \leftarrow N-1, \dots, 0$ **do**
  $P_{n+1} b_n \leftarrow L_{n+1,22} \cdot L_{n+1,22}^T \cdot b_n + p_{n+1}$ ▷ trmv & trmv
  $\begin{bmatrix} l_n \\ p_n \end{bmatrix} \leftarrow \begin{bmatrix} r_n \\ q_n \end{bmatrix} + \begin{bmatrix} B_n^T \\ A_n^T \end{bmatrix} \cdot (P_{n+1} b_n)$ ▷ gemv
  $\begin{bmatrix} l_n \\ p_n \end{bmatrix} \leftarrow \begin{bmatrix} L_{n,11}^{-1} l_n \\ p_n - L_{n,21} \cdot L_{n,11}^{-1} l_n \end{bmatrix}$ ▷ trsv & gemv
end for
if $n_x[0] > 0$ then
  $x_0 \leftarrow -L_{0,22}^{-T} \cdot L_{0,22}^{-1} \cdot p_0$ ▷ trsv
end if
for $n \leftarrow 0, \dots, N$ do
  $u_n \leftarrow -(L_{n,11}^T)^{-1} (L_{n,21}^T \cdot x_n + l_n)$ ▷ gemv & trsv
  $x_{n+1} \leftarrow \begin{bmatrix} B_n & A_n \end{bmatrix} \cdot \begin{bmatrix} u_n \\ x_n \end{bmatrix} + b_n$ ▷ gemv
  $\lambda_{n+1} \leftarrow L_{n+1,22} \cdot L_{n+1,22}^T \cdot x_{n+1} + p_{n+1}$ ▷ trmv & trmv
end for

Considering only terms cubic in the stage variable sizes, the cost of Algorithm 2 is identical to the cost of Algorithm 1.

Considering only terms quadratic in the stage variable sizes, and assuming that the number of states and inputs is constant stage-wise, and equal to n_x and n_u respectively, the cost of Algorithm 3 is

$$N\left(8n_x^2 + 8n_x n_u + 2n_u^2\right)$$

flops in the MPC case, and $2n_x^2$ more in the MHE case. In both the MPC and MHE case, if the Lagrangian multipliers λ_n are not needed, then line 13 can be omitted, saving $N(2n_x^2)$ flops.

8.1.2.2 Implementation using custom linear algebra

The use of custom linear algebra routines can improve the implementation of the square-root backward Riccati recursion algorithm.

Panel-major matrix format The use of the panel-major matrix format as default matrix format in the square-root backward Riccati recursion enables the use of the linear algebra routines proposed in Part I of the thesis. In particular, all matrices passed to the Riccati routine are assumed to be in the panel-major matrix format, and the results of the internal operations are in this matrix format as well. Therefore, the efficient linear algebra routines for embedded optimization can be used without the need to convert the matrices from row-major or column-major format into the panel-major format. For small-scale problems, this notably increases the performance with respect to standard BLAS and LAPACK routines.

Positive semi-definite Hessian The classical backward Riccati recursion requires the matrices $\mathcal{P}$ and $\mathcal{Q}_n$ to be positive semi-definite, while the implementation of the square-root backward Riccati recursion in Section 8.1.2 requires the matrices $\mathcal{P}$ and $\mathcal{Q}_n$ to be strictly positive definite. It is well known that it is possible to compute the Cholesky factor of a positive semi-definite matrix, even if the result in not unique in case of singular matrix. The `potrf` routine proposed in the first part of the thesis can factorize singular matrices.

The square-root backward Riccati recursion as implemented in Algorithm 1 requires the $L_{n,22}$ matrices to be invertible (and therefore it requires the matrices $\mathcal{P}$ and $\mathcal{Q}_n$ to be strictly positive definite). In fact, the matrix $L_{n,22}$ is implicitly used in the computation of $L_{n,32}$ in the Cholesky factorization in line 5, where the operation $L_{n,32} = L_{n,22}^{-1} p_n$ is implicitly performed.

However, the square-root backward Riccati recursion as implemented in Algorithms 2 and 3 works fine also if the matrix $L_{n,22}$ is singular. In fact, in this case the Cholesky factorization $L_{n+1,22}$ of the recursion matrix P_{n+1} is only employed as a numerical trick to speedup the computation of the matrix

$$\begin{bmatrix} Q_n + A_n^T P_{n+1} A_n & S_n^T + A_n^T P_{n+1} B_n \\ S_n + B_n^T P_{n+1} A_n & R_n + B_n^T P_{n+1} B_n \end{bmatrix}$$

and it is not employed in solution of linear systems.

Merging of linear algebra routines Custom routines merging two or more standard BLAS linear algebra routines can be employed in the implementation of Algorithms 1, 2 and 3. In particular, lines 4 and 5 of Algorithms 1 and 2 can be implemented using the single routine `syrk_potrf` proposed in Section 3.3.3. Similarly, line 11 of Algorithm 1, and lines 5 and 11 of Algorithm 3 can be implemented using the single routine `trsv` proposed in Section 4.3.3. The use of merged linear algebra routines gives better computational performance, especially in case of small problems, by reducing the number of calls to linear algebra kernels and increasing the size of the processed matrices and vectors.

8.1.2.3 Relation to array algorithms

The square-root backward Riccati recursion presented above is analogue to the so called array algorithms, but with some important differences.

Array algorithm are widely used for their excellent numerical properties, being employed in the implementation of Kalman filter [55] and Moving Horizon Estimation (MHE) [46]. They propagate a square-root of the recursion matrix P_n, preserving its symmetry and positive-definiteness. The square-root is generally computed using QR factorization. This gives them excellent numerical stability, since the worse-conditioned normal matrix is never formed explicitly.

The QR factorization of a full-rank squared matrix A is $A = Q \cdot R$, with Q orthogonal matrix and R upper-triangular matrix. If all R diagonal elements are required to be positive, then the factorization is unique. The R matrix can be computed using Householder reflections at a cost of about $\frac{4}{3}n^3$ flops, where n is the size of the A matrix.

The uniqueness of the factorization implies that R is equal to the upper Cholesky factor the matrix $A^T \cdot A$. The cost of this algorithm is again $\frac{4}{3}n^3$ flops (n^3 due to the symmetric rank-n update and $\frac{1}{3}n^3$ due to the Cholesky factorization), but it is numerically less stable, since the normal matrix $A^T \cdot A$ is explicitly formed.

If the Cholesky factor R of the matrix $B + A^T \cdot A = R^T \cdot R$ is needed, the cost of the two algorithms is no longer the same. The algorithm based on symmetric rank-n update and Cholesky factorization requires $\frac{4}{3}$ flops also in this case. However, the algorithm based on the QR factorization is more expensive. In fact, R is obtained from the QR factorization of the 'array' matrix

$$\begin{bmatrix} A \\ B^{1/2} \end{bmatrix} = Q \cdot R$$

at the cost of $2(2n)n^2 - \frac{2}{3}n^3 = \frac{10}{3}n^3$ flops. This cost can reduced to $2n^3$ using a custom QR factorization to exploit the fact that $B^{1/2}$ is upper-triangular. If the Cholesky factor $B^{1/2}$ has to be computed, there are further $\frac{1}{3}n^3$ flops.

The proposed square-root backward Riccati recursion algorithm shares with array algorithms the fact that the recursion matrix is the Cholesky factor of P_n, thus automatically enforcing symmetry. The flop count of the algorithm is lower that the equivalent array algorithm (and of the classical backward Riccati recursion as well). However, this comes at the cost of explicitly forming the normal matrix, characterized by a larger condition number.

8.2 Forward Schur-complement recursion

The forward Schur-complement recursion exploits the stage-wise structure of the KKT matrix of the linear MPC (7.4) or of the MHE problem (7.9). This recursion can be seen as a recursive stage-wise factorization of the KKT matrix, where the factorization begins at the top-left corner of the KKT matrix.

Therefore this recursion is naturally suited to solve the linear MHE problem, since it can propagate information starting from the first stage. Traditionally, the linear MHE problem has been solved using a forward Riccati recursion, where each stage of the recursion is analogue to the Covariance Kalman Filter. On the contrary, each stage of the the forward Schur-complement recursion is analogue to the Information Filter formulation of the Kalman filter [14, 65]. The main advantage of the use of the forward Schur-complement recursion over the forward Riccati recursion is its generality (it can deal with the more general MHE problem formulation (7.6), whereas the classical forward Riccati recursion deals with the traditional MHE problem formulation (7.5)). Therefore, the forward Schur-complement recursion can be used to solve the unconstrained QPs arising in constrained and non-linear MHE in a straightforward way.

Furthermore, since the forward Schur-complement recursion is a general recursion of the KKT matrix of the linear OCP, it can also be employed to solve

MPC problems, even if it requires some care in this case. The fact that the factorization starts at the first stage distinguish the forward Schur-complement recursion from the backward Riccati recursion, that begins the factorization at the last stage, and that is therefore naturally suited to solve the linear MPC problem. The forward Schur-complement recursion does not distinguish between state and input variables, and therefore it can be seen as more general than the backward Riccati recursion.

In general, the forward Schur complement recursion requires a slightly larger number of flops compared to the backward Riccati recursion. However, in case of diagonal Hessian, the computational complexity of the forward Schur-complement recursion can be slightly reduced, and made linear in the number of inputs n_u. Furthermore, the recursion can directly and cheaply handle additional equality constraints on the last stage.

8.2.1 Derivation

This section contains the derivation of the forward Schur-complement recursion as a recursive stage-wise factorization of the KKT matrix. As any recursive algorithm, there are two key ingredients: the recursive step and the handling of the end-of-recursion case (plus a starting step in the MPC case). Furthermore, two cases are distinguished: dense and diagonal Hessian.

8.2.1.1 Starting step

The first step is different in the MPC and in the MHE cases. In particular, in the MHE case the first step is identical to the general recursive step, and therefore it is not presented here. On the contrary, the MPC case requires some care, since the Schur complement leads to a singular matrix.

MPC case In the MPC case, the top-left corner of the KKT matrix looks like

$$\left[\begin{array}{cc|cc} R_0 & B_0^T & & \\ B_0 & & -I & \\ \hline & -I & Q_1 & \dots \\ & & \vdots & \ddots \end{array}\right] \left[\begin{array}{c} u_0 \\ \lambda_0 \\ \hline x_1 \\ \vdots \end{array}\right] = - \left[\begin{array}{c} \tilde{r}_0 \\ \tilde{b}_0 \\ \hline q_1 \\ \vdots \end{array}\right].$$

Since the matrix R_0 is assumed to be positive definite and therefore invertible, it is possible to eliminate the variable u_0 and the first equation by means of the

Schur-complement, obtaining

$$\left[\begin{array}{c|cc} -B_0R_0^{-1}B_0^T & -I & \\ \hline -I & Q_1 & \dots \\ & \vdots & \ddots \end{array}\right] \left[\begin{array}{c} \lambda_0 \\ \hline x_1 \\ \vdots \end{array}\right] = - \left[\begin{array}{c} \tilde{b}_0 - B_0R_0^{-1}\tilde{r}_0 \\ \hline q_1 \\ \vdots \end{array}\right].$$

Since the matrix R_0 is assumed to be invertible, the Schur-complement matrix $B_0R_0^{-1}B_0^T$ is invertible if and only if the matrix B_0 has full row rank. In the case $n_x > n_u$ (common in control), the Schur-complement matrix can not be invertible.

One way to proceed is the use of regularization, that has the side effect that the accuracy of the solution is in general lower than using a backward Riccati recursion. Numerical evidence shows that a good value for the regularization parameter ε is often the square root of machine epsilon. The accuracy of the solution is then on the order of the square root of machine epsilon.

An alternative approach is the use of partial condensing (see Chapter 10), that may be employed to increase the input vector size by condensing together several stages. In that case, the matrix B_0 is the controllability matrix, that has full rank if the system is controllable.

In any case, defined

$$P_1^{-1} = B_0R_0^{-1}B_0^T + \varepsilon I$$

for some $\varepsilon \geq 0$ such that the matrix P_1 is invertible, it is possible to eliminate the variable λ_0 and the first equation by explicitly computing the inverse P_1 of the matrix P_1^{-1},

$$\begin{bmatrix} Q_1 + P_1 & \dots \\ \vdots & \ddots \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \end{bmatrix} = - \begin{bmatrix} q_1 - P_1(\tilde{b}_0 - B_0R_0^{-1}\tilde{r}_0) \\ \vdots \end{bmatrix}$$

that can be rewritten in a more compact form as

$$\begin{bmatrix} \Sigma_1 & \dots \\ \vdots & \ddots \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \end{bmatrix} = - \begin{bmatrix} \sigma_1 \\ \vdots \end{bmatrix}.$$

The factorization can continue at the following stage, that has the same form as the general stage, provided that the matrices Q_1 and q_1 are replaced by the updated matrices Σ_1 and σ_1.

8.2.1.2 Recursive step

The recursive step is analogue for MPC and MHE problems. However, two cases will be distinguished, for diagonal and dense Hessian of the cost function.

Diagonal Hessian case The following derivation will assume that the matrices Q_n and R_n are diagonal and that the matrices S_n are zero. This can be exploited to reduce the computational cost of the algorithm. The fact that the matrices S_n, $n = 0, \dots, N-1$ are zero allows for some analogy with the Covariance Kalman filter algorithm for the MHE case.

The matrices Q_n and R_n are further assumed to be positive definite, and the matrices $\begin{bmatrix} A_n & B_n \end{bmatrix}$ are assumed to have full row-rank. These are sufficient conditions to guarantee the invertibility of all matrices in the recursive step. In general, however, these are not necessary conditions.

In the MHE case, recalling the conversion between the traditional MHE formulation (7.5) and the more general MHE formulation (7.6), the matrix Q_0 can be decomposed as

$$Q_0 = \hat{Q}_0 + P_0 = C_0^T \widetilde{R}_0^{-1} C_0 + \widetilde{P}_0^{-1}$$

where $\hat{Q}_0$ is in the same form as the other matrices Q_n, $n > 0$, in the cost function, and the matrix P_0 can be interpreted as the information matrix relative to the initial state prediction $\bar{x}_0$. Similarly, the vector q_0 can be decomposed as

$$q_0 = \hat{q}_0 + p_0 = C_0^T (\bar{v}_0 - y_0) - \widetilde{P}_0^{-1} \bar{x}_0.$$

These expressions are useful to highlight the contribution of the initial state prediction and relative information matrix to the cost function expression, and they will be retained through the recursion.

By introducing the definitions

$$\Sigma_0 = Q_0, \quad \sigma_0 = q_0$$

the top-left corner of the KKT matrix is

$$\left[\begin{array}{ccc|ccc} \Sigma_0 & & A_0^T & & & \\ & R_0 & B_0^T & & & \\ A_0 & B_0 & & -I & & \\ \hline & & -I & Q_1 & \dots & \\ & & & \vdots & \ddots & \end{array}\right] \left[\begin{array}{c} x_0 \\ u_0 \\ \lambda_0 \\ \hline x_1 \\ \vdots \end{array}\right] = - \left[\begin{array}{c} \sigma_0 \\ r_0 \\ b_0 \\ \hline q_1 \\ \vdots \end{array}\right]. \tag{8.10}$$

In general, the matrices Σ_n are assumed to be dense (since at the general stage they are the sum of a diagonal matrix Q_n with a generally dense matrix P_n). Since the matrix Σ_0 is invertible by hypothesis, the variable x_0 can be eliminated using the Schur complement of Σ_0, obtaining

$$\left[\begin{array}{cc|cc} R_0 & B_0^T & & \\ B_0 & -A_0\Sigma_0^{-1}A_0^T & -I & \\ \hline & -I & Q_1 & \dots \\ & & \vdots & \ddots \end{array}\right] \left[\begin{array}{c} u_0 \\ \lambda_0 \\ \hline x_1 \\ \vdots \end{array}\right] = - \left[\begin{array}{c} r_0 \\ b_0 - A_0\Sigma_0^{-1}\sigma_0 \\ \hline q_1 \\ \vdots \end{array}\right].$$

In the MHE case, as a comparison with the Covariance Kalman Filter, notice that by using the matrix inversion lemma, the vector

$$\begin{aligned} \hat{x}_0 &= -\Sigma_0^{-1}\sigma_0 = \\ &= (C_0^T\widetilde{R}_0^{-1}C_0 + \widetilde{P}_0^{-1})^{-1}(C_0^T\widetilde{R}_0^{-1}(y_0 - \bar{v}_0) + \widetilde{P}_0^{-1}\bar{x}_0) = \\ &= (\widetilde{P}_0 - \widetilde{P}_0C_0^T(\widetilde{R}_0 + C_0\widetilde{P}_0C_0^T)^{-1}C_0\widetilde{P}_0)(C_0^T\widetilde{R}_0^{-1}(y_0 - \bar{v}_0) + \widetilde{P}_0^{-1}\bar{x}_0) = \\ &= \bar{x}_0 - \widetilde{P}_0C_0^T(\widetilde{R}_0 + C_0\widetilde{P}_0C_0^T)^{-1}(C\bar{x}_0 + \bar{v}_0 - y_0) \end{aligned}$$

can be interpreted as the state estimation using measurements up to stage 0 in the Kalman filter, and Σ_0 as the relative information matrix. However, $\hat{x}_0$ is not computed explicitly in the information-filter recursion.

Similarly, since the matrix R_0 is invertible by hypothesis, the variable u_0 can be eliminated, obtaining

$$\left[\begin{array}{c|cc} -A_0\Sigma_0^{-1}A_0^T - B_0R_0^{-1}B_0^T & -I & \\ \hline -I & Q_1 & \dots \\ & \vdots & \ddots \end{array}\right] \left[\begin{array}{c} \lambda_0 \\ \hline x_1 \\ \vdots \end{array}\right] = \left[\begin{array}{c} -b_0 + A_0\Sigma_0^{-1}\sigma_0 + B_0R_0^{-1}r_0 \\ \hline -q_1 \\ \vdots \end{array}\right].$$

Since the matrix R_0 is assumed to be diagonal, the computation of the inverse R_0^{-1} is a cheap operation. Finally, since the matrix $P_1^{-1} = A_0\Sigma_0^{-1}A_0^T + B_0R_0^{-1}B_0^T$ is invertible (due to the fact that Σ_0 and R_0 are invertible, and the matrix $\begin{bmatrix} A_0 & B_0 \end{bmatrix}$ has full row rank), the variable λ_0 can be eliminated by explicitly computing the inverse P_1 of the matrix P_1^{-1}, obtaining

$$(Q_1 + P_1)\,x_1 = -q_1 - P_1(-b_0 + A_0\Sigma_0^{-1}\sigma_0 + B_0R_0^{-1}r_0) = -q_1 + P_1\bar{x}_1 = -q_1 - p_1,$$

that can be rewritten in the more compact form

$$\Sigma_1 x_1 = -\sigma_1. \tag{8.11}$$

In general, the matrix Σ_1 is dense, since in general the matrix P_1 is dense.

In the MHE case, as a comparison with the Covariance Kalman Filter, notice that the vector $\bar{x}_1$ can be interpreted as the one-step-ahead state prediction in the Kalman filter given the measurements up to stage 0, and P_1 is the relative information matrix. Furthermore, notice that the matrix $P_1^{-1} = \widetilde{P}_1$ has the form (by using the matrix inversion lemma to compute Σ_0^{-1})

$$\begin{aligned}\widetilde{P}_1 = P_1^{-1} &= B_0 R_0^{-1} B_0^T + A_0 \Sigma_0^{-1} A_0^T = B_0 \widetilde{Q}_0 B_0^T + A_0 (C_0^T \widetilde{R}_0^{-1} C_0 + \widetilde{P}_0^{-1}) A_0^T = \\ &= B_0 \widetilde{Q}_0 B_0^T + A_0 \widetilde{P}_0 A_0^T - A_0 \widetilde{P}_0 C_0^T (\widetilde{R}_0 + C_0 \widetilde{P}_0 C_0^T)^{-1} C_0 \widetilde{P}_0 A_0^T\end{aligned}$$

that is the classical expression of the forward Riccati recursion of the Covariance Kalman Filter. However, in the forward Schur-complement recursion, the inversion of the matrix Σ_0 is computed explicitly (similarly to the information Kalman filter) instead of by means of the matrix inversion lemma (as in the covariance Kalman filter).

Equation (8.11) closes the recursion, since now the top-left corner of the KKT matrix is

$$\left[\begin{array}{ccc|ccc} \Sigma_1 & & A_1^T & & & \\ & R_1 & B_1^T & & & \\ A_1 & B_1 & & -I & & \\ \hline & & -I & Q_2 & \dots & \\ & & & \vdots & \ddots & \end{array}\right] \left[\begin{array}{c} x_1 \\ u_1 \\ \lambda_1 \\ \hline x_2 \\ \vdots \end{array}\right] = \left[\begin{array}{c} -\sigma_1 \\ -r_1 \\ -b_1 \\ \hline -q_2 \\ \vdots \end{array}\right].$$

that is in the same form as (8.10). The recursion can therefore be repeated at the following stage.

Dense Hessian case The following derivation will not make hypothesis on the sparsity structure of the matrices Q_n, R_n and S_n: therefore they are assumed to be dense. In the MHE case, no comparisons is made with the covariance Kalman Filter.

The matrices $\begin{bmatrix} Q_n & S_n^T \\ S_n & R_n \end{bmatrix}$ and Q_N are assumed to be positive definite, and the matrices $\begin{bmatrix} A_n & B_n \end{bmatrix}$ are assumed to have full row-rank. These are sufficient conditions to guarantee the invertibility of all matrices in the recursion. In general, however, these are not necessary conditions.

By introducing the definitions

$$\Sigma_0 = Q_0, \quad \sigma_0 = q_0$$

the top-left corner of the KKT matrix is

$$\left[\begin{array}{ccc|ccc} \Sigma_0 & S_0^T & A_0^T & & & \\ S_0 & R_0 & B_0^T & & & \\ A_0 & B_0 & & -I & & \\ \hline & & -I & Q_1 & \dots & \\ & & & \vdots & \ddots & \end{array}\right] \left[\begin{array}{c} x_0 \\ u_0 \\ \lambda_0 \\ \hline x_1 \\ \vdots \end{array}\right] = - \left[\begin{array}{c} \sigma_0 \\ r_0 \\ b_0 \\ \hline q_1 \\ \vdots \end{array}\right]. \tag{8.12}$$

Since the matrix $\begin{bmatrix} \Sigma_0 & S_0^T \\ S_0 & R_0 \end{bmatrix}$ is invertible by hypothesis, the variables x_0 and u_0 can be eliminated using the Schur complement of $\begin{bmatrix} \Sigma_0 & S_0^T \\ S_0 & R_0 \end{bmatrix}$, obtaining

$$\left[\begin{array}{cc|ccc} - \begin{bmatrix} A_0 & B_0 \end{bmatrix} \begin{bmatrix} \Sigma_0 & S_0^T \\ S_0 & R_0 \end{bmatrix}^{-1} \begin{bmatrix} A_0^T \\ B_0^T \end{bmatrix} & & -I & & \\ \hline & -I & Q_1 & \dots & \\ & & \vdots & \ddots & \end{array}\right] \left[\begin{array}{c} \lambda_0 \\ \hline x_1 \\ \vdots \end{array}\right] =$$

$$= - \left[\begin{array}{c} b_0 - \begin{bmatrix} A_0 & B_0 \end{bmatrix} \begin{bmatrix} \Sigma_0 & S_0^T \\ S_0 & R_0 \end{bmatrix}^{-1} \begin{bmatrix} \sigma_0 \\ r_0 \end{bmatrix} \\ \hline q_1 \\ \vdots \end{array}\right].$$

Finally, since the matrix $P_1^{-1} = \begin{bmatrix} A_0 & B_0 \end{bmatrix} \begin{bmatrix} \Sigma_0 & S_0^T \\ S_0 & R_0 \end{bmatrix}^{-1} \begin{bmatrix} A_0^T \\ B_0^T \end{bmatrix}$ is invertible (due to the fact that the matrix $\begin{bmatrix} \Sigma_0 & S_0^T \\ S_0 & R_0 \end{bmatrix}$ is invertible, and the matrix $\begin{bmatrix} A_0 & B_0 \end{bmatrix}$ has full row rank), the variable λ_0 can be eliminated, obtaining

$$\begin{aligned} (Q_1 + P_1)\, x_1 &= - \left(q_1 - P_1 \left(b_0 - \begin{bmatrix} A_0 & B_0 \end{bmatrix} \begin{bmatrix} \Sigma_0 & S_0^T \\ S_0 & R_0 \end{bmatrix}^{-1} \begin{bmatrix} \sigma_0 \\ r_0 \end{bmatrix} \right) \right) = \\ &= - \left(q_1 + p_1 \right), \end{aligned}$$

that can be rewritten in the more compact form

$$\Sigma_1 x_1 = -\sigma_1.$$

This closes the recursion, since now the top-left corner of the KKT matrix is

$$\left[\begin{array}{ccc|ccc} \Sigma_1 & S_1^T & A_1^T & & & \\ S_1 & R_1 & B_1^T & & & \\ A_1 & B_1 & & -I & & \\ \hline & & -I & Q_2 & \dots & \\ & & & \vdots & \ddots & \end{array}\right] \left[\begin{array}{c} x_1 \\ u_1 \\ \lambda_1 \\ \hline x_2 \\ \vdots \end{array}\right] = - \left[\begin{array}{c} \sigma_1 \\ r_1 \\ b_1 \\ \hline q_2 \\ \vdots \end{array}\right].$$

that is in the same form as (8.12). The recursion can therefore be repeated at the following stage.

8.2.1.3 End of recursion

At the end of the recursion, two cases are distinguished, based on the presence or not of equality constraints on the last stage.

Case $n_d = 0$ If there are no equality constraints on the last stage variables, the last stage looks like

$$\Sigma_N x_N = -\sigma_N.$$

This linear system of equations can be solved by means of Cholesky factorization of the positive definite matrix Σ_N and forward-backward substitutions.

In the MHE case, Σ_N is the information matrix of the estimate x_N, and it is available at no extra computational cost.

Case $n_d > 0$ If there are equality constraints on the last stage variables, the last stage looks like

$$\begin{bmatrix} \Sigma_N & D_N^T \\ D_N & \end{bmatrix} \begin{bmatrix} x_N \\ \lambda_N \end{bmatrix} = - \begin{bmatrix} \sigma_N \\ d_N \end{bmatrix}$$

This can be solved by computing the Schur complement matrix of Σ_N, that gives the linear system of equations

$$\left[D_N \Sigma_N^{-1} D_N^T\right] \left[\lambda_N\right] = \left[d_N - D_N \Sigma_N^{-1} \sigma_N\right]$$

The matrix $D_N \Sigma_N^{-1} D_N^T$ is positive definite in the hypothesis that the matrix Σ_N is positive definite and that the constraint matrix D_N has full row rank. In that hypothesis, the system can be easily solved for λ_N by means of Cholesky factorization followed by forward-backward substitutions.

Once the value of λ_N is known, the value of x_N can be computed as

$$x_N = \Sigma_N^{-1}(-\sigma_N - D_N^T \lambda_N).$$

In the MHE case, the information matrix of the estimate in the null-space can be computed as

$$\Sigma_{Z,N} = Z^T \Sigma_N Z$$

where Z is a null-space matrix of D_N [67]. If the D matrix fixes the value of some of the states (i.e. it consists of rows from an identity matrix), then a suitable Z matrix is trivially a collection of the columns from an identity matrix corresponding to the free states, and $E_{Z,2}$ reduces to the elements of E_2 corresponding to the free states.

8.2.2 Implementation

In this section, implementation of the forward Schur-complement recursion is presented. Firstly the algorithm is presented using standard BLAS and LAPACK routines, and the computational complexity as number of flops is derived for the algorithm in both cases of dense and diagonal Hessian of the cost function. Then the algorithm is presented using the custom linear algebra routines proposed it Part I of the thesis.

8.2.2.1 Implementation using BLAS and LAPACK

The forward Schur-complement recursion can be implemented using the standard BLAS and LAPACK routines. The algorithm is summarized in Algorithm 4, where the name of the BLAS and LAPACK routines employed in the implementation appreas as a comment.

In the case square-root backward Riccati recursion, the key operation is the computation of $\mathcal{Q}+\mathcal{A}^T\cdot\mathcal{P}\cdot\mathcal{A}$, where $\mathcal{Q}$ is a positive semi-definite matrix. If all matrices $\mathcal{A}$, $\mathcal{P}$ and $\mathcal{Q}$ have size n, then the most efficient way to compute this operation is

$$\mathcal{Q}+\mathcal{A}^T\cdot\mathcal{P}\cdot\mathcal{A}=\mathcal{Q}+\mathcal{A}^T\cdot(\mathcal{L}\cdot\mathcal{L}^T)\cdot\mathcal{A}=\mathcal{Q}+(\mathcal{A}^T\cdot\mathcal{L})\cdot(\mathcal{A}^T\cdot\mathcal{L})^T \qquad (8.13)$$

where $\mathcal{L}$ is the lower Cholesky factor of $\mathcal{P}$. Using specialized BLAS routines, the cost of this operation is $\frac{1}{3}n^3$ (`potrf`) + n^3 (`trmm`) + n^3 (`syrk`) = $\frac{7}{3}n^3$ flops.

In the case of the forward Schur-complement recursion, the key operation is the computation of $\mathcal{Q}+\mathcal{A}\cdot\mathcal{P}^{-1}\cdot\mathcal{A}^T$, where $\mathcal{Q}$ is a positive definite matrix. Despite the presence of a matrix inversion, this operation can be computed in the exact same number of flops as the operation in (8.13). In fact, the matrix inversion is computed implicitly, as

$$\mathcal{Q}+\mathcal{A}\cdot\mathcal{P}^{-1}\cdot\mathcal{A}^T=\mathcal{Q}+\mathcal{A}\cdot(\mathcal{L}\cdot\mathcal{L}^T)^{-1}\cdot\mathcal{A}^T=\mathcal{Q}+(\mathcal{A}\cdot\mathcal{L}^{-T})\cdot(\mathcal{A}\cdot\mathcal{L}^{-T})^T \quad (8.14)$$

where again $\mathcal{L}$ is the lower Cholesky factor of $\mathcal{P}$. Since the matrix $\mathcal{L}$ is triangular, the operation $\mathcal{A}\cdot\mathcal{L}^{-T}$ can be computed efficiently using the routine `trsm`

Algorithm 4 Forward Schur-complement recursion - factorization - dense Hessian

$\mathcal{Q} \leftarrow \begin{bmatrix} Q_0 & 0 \\ S_0 & R_0 \end{bmatrix}$
$\mathcal{A} \leftarrow \begin{bmatrix} A_0 & B_0 \end{bmatrix}$
$\mathcal{L}_0 \leftarrow Q^{1/2}$ ▷ potrf
$\mathcal{AL}_0 \leftarrow \mathcal{A} \cdot \mathcal{L}^{-T}$ ▷ trsm
$P_{inv} \leftarrow \mathcal{AL}_0 \cdot \mathcal{AL}_0^T$ ▷ syrk
$L \leftarrow P_{inv}^{1/2}$ ▷ potrf
$U_1 \leftarrow L^{-T}$ ▷ trtri
for $n \leftarrow 1, \ldots, N-1$ do
    $\Sigma \leftarrow Q_n + U_n \cdot U_n^T$ ▷ lauum
    $\mathcal{Q} \leftarrow \begin{bmatrix} \Sigma & 0 \\ S_n & R_n \end{bmatrix}$
    $\mathcal{A} \leftarrow \begin{bmatrix} A_n & B_n \end{bmatrix}$
    $\mathcal{L}_n \leftarrow Q^{1/2}$ ▷ potrf
    $\mathcal{AL}_n \leftarrow \mathcal{A} \cdot \mathcal{L}^{-T}$ ▷ trsm
    $P_{inv} \leftarrow \mathcal{AL}_n \cdot \mathcal{AL}_n^T$ ▷ syrk
    $L \leftarrow P_{inv}^{1/2}$ ▷ potrf
    $U_{n+1} \leftarrow L^{-T}$ ▷ trtri
end for
$\Sigma \leftarrow Q_N + U_N \cdot U_N^T$ ▷ lauum
$\mathcal{L}_N \leftarrow \Sigma^{1/2}$ ▷ potrf
if $n_d > 0$ then
    $\mathcal{A} \leftarrow \begin{bmatrix} D_N \end{bmatrix}$
    $\mathcal{AL}_N \leftarrow \mathcal{A} \cdot \mathcal{L}^{-T}$ ▷ trsm
    $P_{inv} \leftarrow \mathcal{AL}_N \cdot \mathcal{AL}_N^T$ ▷ syrk
    $L_N \leftarrow P_{inv}^{1/2}$ ▷ potrf
end if

to solve a triangular system of linear equations with matrix RHS. Using specialized BLAS routines, the cost of this operation is $\frac{1}{3}n^3$ (potrf) + n^3 (trsm) + n^3 (syrk) = $\frac{7}{3}n^3$ flops. This makes the forward Schur-complement recursion competitive with respect to the square-root backward Riccati recursion.

The following analysis of the computational cost includes only terms that are cubic in the stage variable sizes, and assumes the number of states and inputs to be constant stage-wise, and equal to n_x and n_u respectively. In case of dense Hessian of the cost function, the computational cost of Algorithm 4 is of

$$N(\tfrac{10}{3}n_x^3 + 4n_x^2 n_u + 2n_x n_u^2 + \tfrac{1}{3}n_u^3) + \tfrac{1}{3}n_x^3 + n_d n_x^2 + n_d^2 n_x + \tfrac{1}{3}n_d^3$$

flops for the MHE case, and $\frac{4}{3}n_x^3 + 3n_x^2 n_u + n_x n_u^2$ flops less for the MPC case.

If the Hessian of the cost function is sparse, the computational cost can be reduced, and this is true also stage-wise (i.e. if the Hessian matrices are sparse only at some stages). Namely:

- a zero S_n matrix can be exploited at all stages, to reduce the computational cost by $3n_x^2 n_u + n_x n_u^2$ flops per stage.
- if S_n is a zero matrix, a diagonal R_n matrix can be exploited at all stages to reduce the computational cost by additional $n_x n_u^2 + \frac{1}{3}n_u^3$ per stage.
- a diagonal Q_n matrix can be exploited only at the first stage, since afterwards it is updated with a dense matrix, as in lines 9 and 18 of Algorithm 4. In this case, the cost of the first iteration can be reduced by additional $\frac{4}{3}n_x^3$ flops if $S_0 = 0$, or additional $\frac{4}{3}n_x^3 + n_x^2 n_u$ flops if $S_0 \neq 0$.

Notice that BLAS and LAPACK do not have support for diagonal matrices, and therefore custom routines need to be employed for that. In conclusion, if the Hessian of the cost function is diagonal at all stages, the computational cost of Algorithm 4 is of

$$N(\tfrac{10}{3}n_x^3 + n_x^2 n_u) - n_x^3 + n_d n_x^2 + n_d^2 n_x + \tfrac{1}{3}n_d^3$$

flops for the MHE case, and n_x^3 flops less for the MPC case. Therefore in case of diagonal Hessian of the cost function, the computational complexity of the factorization part of the forward Schur-complement recursion is linear in n_u.

The algorithm for the solution of the KKT system given the factorization of the KKT matrix is presented in Algorithm 5, where the name of the BLAS and LAPACK routines employed in the implementation appears as a comment.

Algorithm 5 consists of forward and backward substitutions. Again, triangular matrices are exploited by means of specialized routines. The following analysis of the computational cost includes only terms that are quadratic in the stage variable sizes, and assumes the number of states and inputs to be constant stage-wise, and equal to n_x and n_u respectively. In case of dense Hessian of the cost function, the computational cost of Algorithm 5 is of

$$N\left(10n_x^2 + 8n_x n_u + 2n_u^2\right) + 2n_x^2 + 4n_x n_d + 2n_d^2$$

flops for the MHE case, and $6n_x^2 + 4n_x n_u$ flops less for the MPC case.

If the Hessian of the cost function is sparse, the computational cost can be reduced, and this is true also stage-wise (i.e. if the Hessian matrices are sparse only at some stages). Namely:

Algorithm 5 Forward Schur-complement recursion - solution - dense Hessian

1: $\begin{bmatrix}\bar{q}_0\\ \bar{r}_0\end{bmatrix} \leftarrow \mathcal{L}_0^{-1} \cdot \begin{bmatrix}\sigma_0\\ r_0\end{bmatrix}$ ▷ trsv
2: $\bar{b}_0 \leftarrow b_0 - \mathcal{AL}_0 \cdot \begin{bmatrix}\bar{q}_0\\ \bar{r}_0\end{bmatrix}$ ▷ gemv
3: for $n \leftarrow 1, \dots, N-1$ do
4: $\quad \sigma_n \leftarrow q_n - U_n \cdot U_n^T \cdot \bar{b}_{n-1}$ ▷ trmv
5: $\quad \begin{bmatrix}\bar{q}_n\\ \bar{r}_n\end{bmatrix} \leftarrow \mathcal{L}_n^{-1} \cdot \begin{bmatrix}\sigma_n\\ r_n\end{bmatrix}$ ▷ trsv
6: $\quad \bar{b}_n \leftarrow b_n - \mathcal{AL}_n \cdot \begin{bmatrix}\bar{q}_n\\ \bar{r}_n\end{bmatrix}$ ▷ gemv
7: end for
8: $\sigma_N \leftarrow q_N - U_N \cdot U_N^T \cdot \bar{x}_N$ ▷ trmv
9: if $n_d = 0$ then
10: $\quad x_N \leftarrow -\mathcal{L}_N^{-T} \cdot \mathcal{L}_N^{-1} \cdot \sigma_N$ ▷ trsv
11: else
12: $\quad \bar{b}_N \leftarrow d_N - \mathcal{AL}_N \cdot \mathcal{L}_N^{-1} \cdot \sigma_N$ ▷ gemv & trsv
13: $\quad \lambda_N \leftarrow L_N^{-T} \cdot L_N^{-1} \cdot \bar{b}_N$ ▷ trsv
14: $\quad x_N \leftarrow \mathcal{L}_N^{-T} \cdot \left(-\sigma_N - \mathcal{AL}_N^T \cdot \lambda_N\right)$ ▷ gemv & trsv
15: end if
16: for $n \leftarrow N-1, \dots, 0$ do
17: $\quad \lambda_n \leftarrow U_n \cdot U_n^T \cdot (\bar{b}_n - x_{n+1})$ ▷ trmv
18: $\quad \begin{bmatrix}x_n\\ u_n\end{bmatrix} \leftarrow \mathcal{L}_n^{-T} \cdot \left(-\begin{bmatrix}\bar{q}_n\\ \bar{r}_n\end{bmatrix} - \mathcal{AL}_n^T \cdot \lambda_n\right)$ ▷ gemv & trsv
19: end for

- a zero S_n matrix can be exploited at all stages, to reduce the computational cost by $4n_x n_u$ flops per stage.
- if S_n is a zero matrix, a diagonal R_n matrix can be exploited at all stages to reduce the computational cost by additional $2n_u^2$ per stage.
- a diagonal Q_n matrix can be exploited only at the first stage, since afterwards it is updated with a dense matrix, as in lines 9 and 18 of Algorithm 4. In this case, the cost of the first iteration can be reduced by additional $2n_x^2$ flops.

Again, BLAS and LAPACK do not have support for diagonal matrices, and therefore custom routines need to be employed for that. In conclusion, if the Hessian of the cost function is diagonal at all stages, the computational cost of Algorithm 4 is of

$$N\left(10n_x^2 + 4n_x n_u\right) + 4n_x n_d + 2n_d^2$$

flops for the MHE case, and $4n_x^2$ flops less for the MPC case. Therefore in case of diagonal Hessian of the cost function, the computational complexity also of the solution part forward Schur-complement recursion is linear in n_u, and therefore so it is the computational complexity of the entire recursion.

8.2.2.2 Implementation using BLAS and LAPACK

The use of custom linear algebra routines can improve the implementation of the forward Schur-complement recursion algorithm.

Panel-major matrix format The use of the panel-major matrix format as the default matrix format in the forward Schur-complement recursion enables the use of the linear algebra routines proposed in Part I ef the thesis. In particular, all matrices passed to the Schur-complement recursion routine are assumed to be in the panel-major matrix format, and the results of the internal operations are in this matrix format as well. Therefore, the efficient linear algebra routines for embedded optimization can be used without the need to convert the matrices from row-major or column-major into the panel-major format. For small-scale problems, this notably increases the computational performance with respect to the use of standard BLAS and LAPACK routines.

Merging of linear algebra routines Custom routines merging two or more standard BLAS or LAPACK linear algebra routines can be employed in the implementation of Algorithms 4 and 5. In particular, lines 3 and 4, 12 and 13, 19 and 22 of Algorithm 4 can be implemented using the single routine `potrf` proposed in Section 3.3.3: this routine merges the standard LAPACK routine `potrf` and the standard BLAS routine `trsm`. Additionally, lines 9 to 12 of Algorithm 4, 18 to 19 can be implemented using the single routine `lauum_potrf` proposed in Section 3.3.3: this routine merges the standard LAPACK routines `lauum` and `potrf` and the standard BLAS routine `trsm`. The `gemv` and `trsv` operations in line 1 and 2, 5 and 6, 12, 14, 18 of Algorithm 5 can be implemented using the single routine `trsv` proposed in Section 4.3.3. The used of merged linear algebra routines gives better computational performance, especially in case of small problems, by reducing the number of calls to linear algebra kernels and increasing the size of processed matrix and vectors.

8.3 Comparison of structure-exploiting factorizations

This section contains the comparison of the implementation of the structure-exploiting recursive factorization algorithms. Several tests are performed: the backward Riccati recursion is compared to the forward Schur-complement recursion (for both full and diagonal Hessian of the cost function) when both recursions are implemented using the custom linear algebra routines in HPMPC. Furthermore, different implementations of each single recursion are compared, namely when implemented employing linear algebra routines in HPMPC or in BLAS and LAPACK libraries (reference BLAS and LAPACK 3.5.0 from Netlib, MKL 11.3 and OpenBLAS 0.2.15). Finally, the tests are performed on both an Intel Ivy-Bridge processor (supporting the AVX ISA) and an Intel Haswell processor (supporting the AVX2 and FMA ISAs). The two test machines have the same memory configuration, namely 8 GB of DDR3/DDR3L memory in dual-channel configuration (for a total data width of 128 bits), running at 1600 MHz, that gives a maximum bandwidth of 25.6 GB/s. Therefore, the difference in performance is solely due to the processors.

The test problem is a LTI MPC test problem, the unconstrained version of the widely-employed mass-spring system [90]. Flush to zero of denormals is enabled: this avoids the large decrease of performance that would otherwise happen when the matrix exponential operation in the discretization of the test matrices produces denormal numbers. The exact nature of the test problem does not affect the computational times, provided that denormals are flushed to zero, and that no Inf or NaN are produced in the computations. Since the complexity of all algorithms is linear in N, the horizon length is fixed to $N = 10$ (or to $N = 100$ in a few tests). The number of states n_x is varied between 4 and 300 in steps of 4 (plus the sizes $n_x = 6$ and $n_x = 10$ to increase the resolution for small-scale systems), and the number of inputs n_u is equal to half of the number of states (and therefore varied between 2 and 150 in steps of 2). In the case of the MPC problem, (at least) the first stage of the forward Schur-complement recursion requires regularization. Therefore, a regularization of $\varepsilon = 10^{-9}$ is employed at all stages in the forward Schur-complement recursion algorithms.

8.3.1 Comparison on Intel Ivy-Bridge micro-architecture

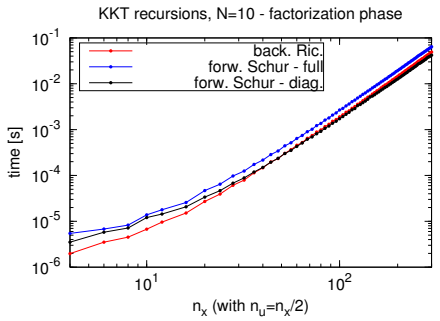
The test processor is the same Intel i7 3520M used for the tests in Sections 3.5.1 and 4.4.1. The Ivy-Bridge micro-architecture supports the AVX ISA, and it can perform a 256-bit-wide multiplication and a 256-bit addition every clock cycle.

8.3.1.1 Comparison of algorithms in HPMPC

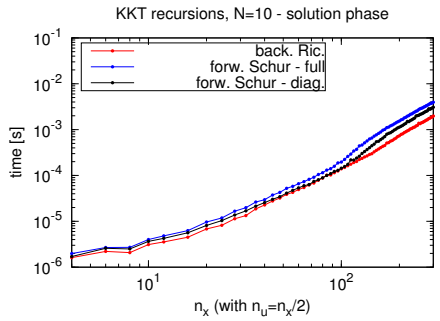
Figure 8.1 contains the results of timing and performance tests for the developed recursive factorizations and solutions.

Figure 8.1a compares the KKT matrix factorization time for an horizon of $N = 10$. For the tested problem sizes, the backward Riccati recursion is always faster than the forward Schur-complement recursion in case of dense Hessian of the cost function. In case of diagonal Hessian of the cost function, for large-scale problems the forward Schur-complement recursion is slightly faster than the backward Riccati recursion. Figure 8.1c compares the computational performance of the KKT matrix factorization routines for an horizon of $N = 10$. The backward Riccati recursion has slightly better performance than the forward Schur-complement recursion (in both cases of dense and diagonal Hessian of the cost function). This is due to the fact that in the forward Schur-complement recursion a larger fraction of flops is due to LAPACK routines (that generally attain a slightly lower computational performance, especially for small matrices, see Section 3.5). Figure 8.1e compares the computational performance of the KKT matrix factorization routines for an horizon of $N = 100$. The figure is very similar to the one for $N = 10$ (Figure 8.1e), with the only difference that the performance of routines (and especially the forward Schur-complement routines) is slightly lower for n_x larger than about 40, with the performance penalty decreasing as n_x increases. The behavior is due to the fact that, for n_x smaller than about 40, all the data structure can fit in L3 cache at once, and therefore the single matrices do not need to be streamed from main memory. For n_x larger than about 40, each single matrix needs to be streamed from main memory, but then it fits in cache when the matrix elements are reused in level 3 BLAS and LAPACK routines. Therefore, for large values of n_x the cost to stream each matrix from memory is amortized over a large number of flops, and the performance is almost indistinguishable from the case $N = 10$.

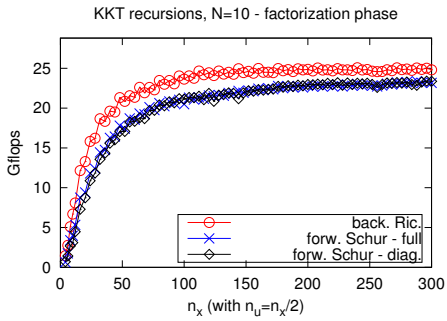
Figure 8.1b compares the KKT system solution time, once the KKT matrix is factorized, for $N = 10$. For the test problem sizes, the backward Riccati recursion is always faster. The difference is small for small problems, but it gets larger for n_x larger than about 100, especially in the case of dense Hessian of the cost function. The performance plots in Figure 8.1d (for $N = 10$) can explain this. The performance is very similar for all routines for small-scale problems, but for x_n larger than about 100, the performance decreases quickly for the forward Schur-complement recursion, especially in the case of dense Hessian of the cost function. A comparison with the performance level of level 2 BLAS routines in Section 4.4 reveals that for n_x smaller than about 100, the performance is about at the same level as the performance when data is streamed from level 3 cache, while for large n_x it settles at a much lower level, since the



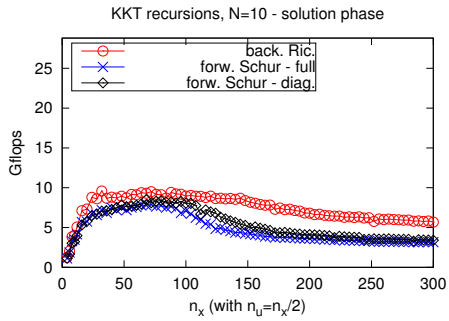
(a) Factorization time.



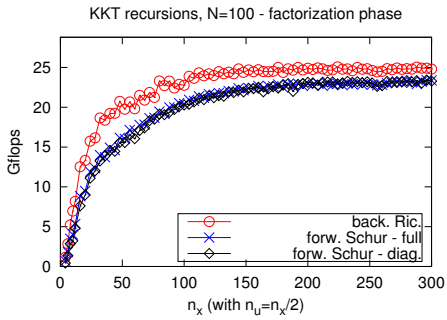
(b) Solution time.



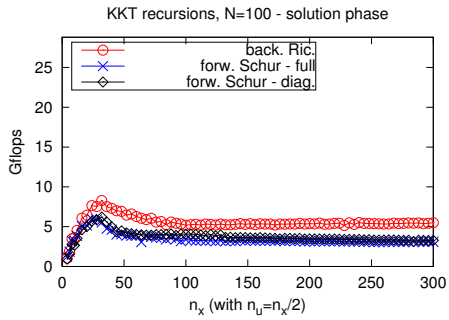
(c) Factorization Gflops.



(d) Solution Gflops.



(e) Factorization Gflops.



(f) Solution Gflops.

Figure 8.1: KKT recursive factorization tests on Intel i7 3520M (Intel Ivy-Bridge micro-architecture, supporting the AVX ISA).

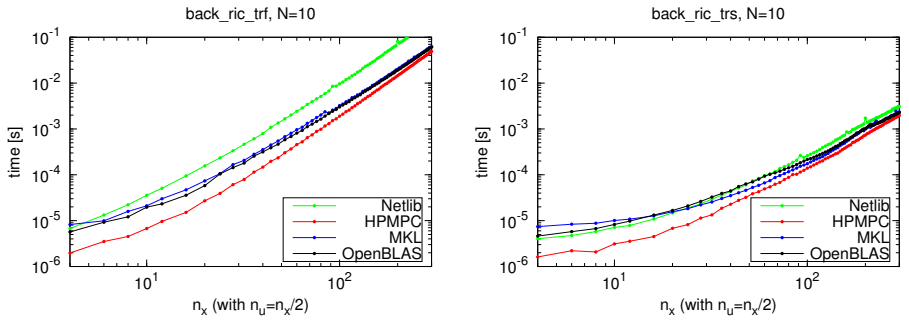
data is streamed from main memory. Figure 8.1d contains the performance plot for $N = 100$. In this case, the performance drops for a much smaller value of n_x. Since in level 2 BLAS routines there is no reuse of matrix elements, once the performance drops due to the streaming of matrix data from main memory, the performance keeps being low as n_x increases. In the tested LTI case, the performance of the backward Riccati recursion solution is slightly higher since the dynamical system matrices are re-used in consecutive iterations, and therefore already present in cache (at least for all tested problem sizes; for values of n_x large enough each such that each single matrix does not fit in L3 cache, the performance drops also for the backward Riccati recursion solution in the LTI case). All other matrices in the backward Riccati recursion solution and all matrices in the forward Schur-complement recursion solution change at each stage, preventing any reuse of matrix data in cache. The same happens for all solvers in case of time-variant test problems.

The KKT system solution routine is implemented using exclusively level 2 BLAS routines, and therefore there is no reuse of matrix elements within the single operations. The same matrices can be reused at consecutive stages (for the dynamical system matrices in case of LTI problems, as in the test), or at the same stage in the backward and forward recursions (in all cases). Therefore, if the problem is small enough such that the entire problem data fits in some cache level, the performance is analogue to the one of the level 2 BLAS routines for matrices fitting in that cache level. As a consequence, if the problem matrices are time-variant, or if the horizon length N is large, the performance drop happens for smaller values of n_x.

8.3.1.2 Comparison of algorithms between HPMPC and optimized BLAS

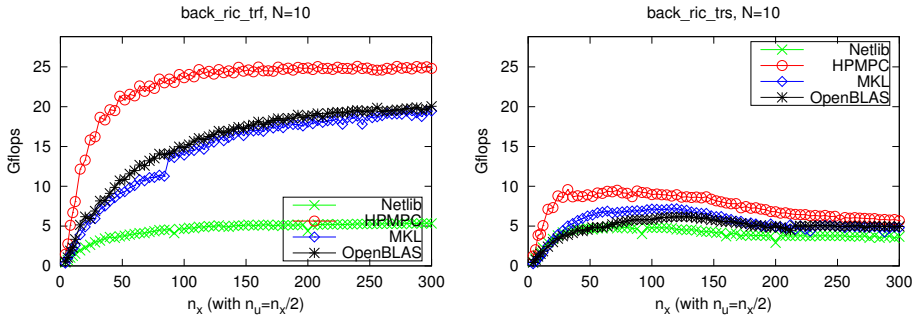
In Figure 8.2, the backward Riccati recursion has been implemented using linear algebra routines from HPMPC or from different BLAS and LAPACK versions.

Figure 8.2a contains the results for the factorization time. For small-scale problems, all BLAS version perform similarly, while HPMPC gives a nice performance boost. For larger-scale problems, the performance advantage of HPMPC over optimized BLAS libraries gets smaller, while it keeps constant to about 5 times as fast as the reference BLAS and LAPACK. Both optimized BLAS libraries (MKL and OpenBLAS) perform very similarly, with OpenBLAS being slightly better than MKL. This is confirmed also by the analysis of the performance plot in Figure 8.2c. The performance of the factorization routine employing the linear algebra routines in HPMPC increases steeply for small matrices, and keeps steady around 80-85% of full FP throughput for larger problem



(a) Backward Riccati recursion factorization time.

(b) Backward Riccati recursion solution time.



(c) Backward Riccati recursion factorization Gflops.

(d) Backward Riccati recursion solution Gflops.

Figure 8.2: Backward Riccati recursion tests on Intel i7 3520M (Intel Ivy-Bridge micro-architecture, supporting the AVX ISA).

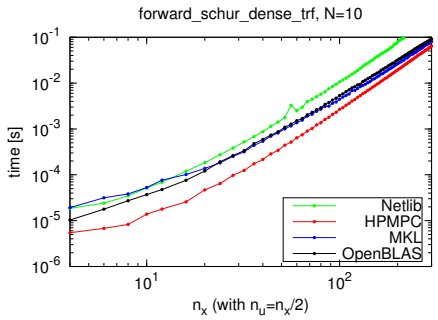
sizes. Employing linear algebra routines from optimized BLAS and LAPACK libraries, the performance increases as the problem size increases. On the contrary, reference linear algebra routines give low performance (about 15-18% of full FP throughput) for all problem sizes.

Figure 8.2b contains the results for the solution time. In this case, the linear algebra routines in HPMPC still give a nice performance boost for small matrices, but as the problem size increases the performance gets similar for all implementations. This is due to the fact that the solution time is bounded by the time needed to stream data from main memory. In the implementation of the solution algorithm, optimized BLAS versions give little advantage over the reference BLAS version. The performance plot in Figure 8.2d confirms this, showing as the performance gets low for all implementations as the problem size increases. Nonetheless, even if partially counter-balanced by a much lower performance, the solution time is still about one order of magnitude lower than the factorization time in case of large-scale problems.

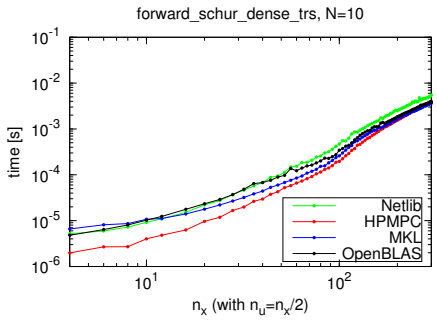
In Figure 8.3, the forward Schur-complement recursion has been implemented using linear algebra routines from HPMPC or from different BLAS and LAPACK versions. Only the case of full Hessian of the cost function is considered.

Figure 8.3a contains the results for the factorization time. Overall, the results are very similar to the ones for the backward Riccati recursion, with the difference that the factorization times are slightly higher. This is partially due to the higher flop count, and partially to the slightly lower performance of the factorization routine, as shown in Figure 8.3c. The lower performance is due to the fact that these routines contains more LAPACK routines (triangular matrix factorizations and inversions), that typically can attain a lower performance. The linear algebra routines in HPMPC still give the best performance, while the use of reference BLAS and LAPACK very low performance. About optimized BLAS versions, this time MKL slightly outperforms OpenBLAS, likely due to the extremely poor performance of the triangular matrix inversion routine in OpenBLAS.

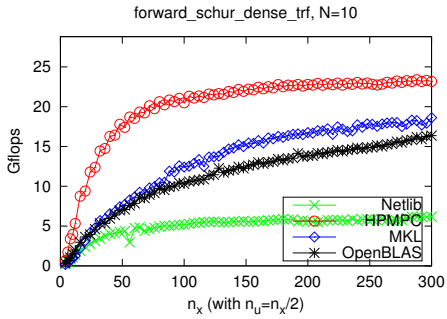
Figure 8.3b contains the results for the solution time. Again, the results are very similar to the ones for the backward Riccati recursion in Figure 8.2b. The linear algebra routines in HPMPC give some performance improvements for small-scale problems, while the performance is very similar for all implementations for large-scale problems.



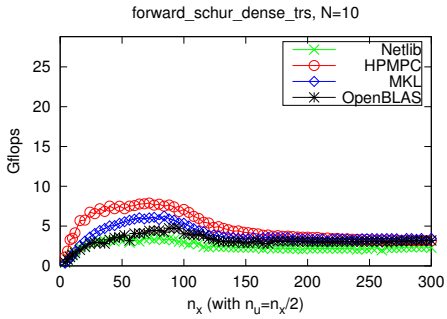
(a) Forward Schur-complement recursion factorization time.



(b) Forward Schur-complement recursion solution time.



(c) Forward Schur-complement recursion factorization Gflops.



(d) Forward Schur-complement recursion solution Gflops.

Figure 8.3: Forward Schur-complement recursion tests on Intel i7 3520M (Intel Ivy-Bridge micro-architecture, supporting the AVX ISA).

8.3.2 Comparison on Intel Haswell micro-architecture

The test processor is the same Intel i7 4800MQ used for the tests in Sections 3.5.2 and 4.4.2. The Haswell micro-architecture supports the AVX2 and FMA ISAs, and it can perform two 256-bit-wide fused-multiplication-addition every clock cycle.

Qualitatively, the results on the Haswell processor are similar than on the Ivy-Bridge processor, with the main differences outlines here.

8.3.2.1 Comparison of algorithms in HPMPC

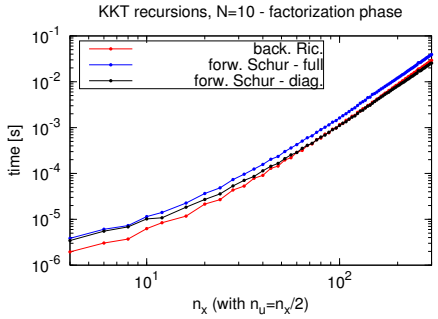
In case of the factorization routines, Figure 8.4a is similar to the Ivy-Bridge counterpart in Figure 8.1a, with the backward Riccati recursion begin up to twice as fast for small-scale problems, and with the diagonal Hessian version of the forward Schur-complement recursion being slightly faster for large-scale problems. Comparing the performance plots (Figures 8.4c and 8.4e), in the case of the Haswell processor the difference in performance between the backward Riccati recursion and the forward Schur-complement recursion is slightly larger, especially in case of long horizon (Figure 8.4e, where a performance drop is clearly visible when the data footprint exceeds L3 cache size. The performance of the forward Schur-complement recursion is more sensitive to cache size due to the larger memory usage.

In case of the solution routines, the main difference between the Ivy-Bridge processor and the Haswell processor is the fact that the drop in performance happens for larger matrix sizes, since the L3 cache size is 6 MB on the Haswell processor and 4 MB on the Ivy-Bridge processor. The performance levels are similar on the two processor, since the performance is mainly due to the bandwidth of L3 cache and of main memory, that is similar for the two processors.

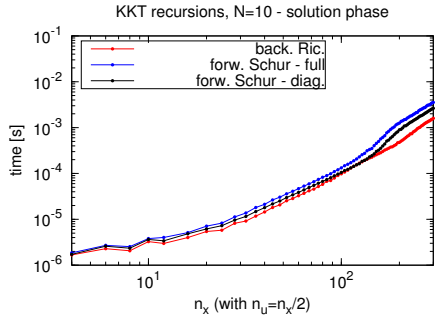
8.3.2.2 Comparison of algorithms between HPMPC and optimized BLAS

In the comparison with other BLAS and LAPACK implementations, there is some noteworthy difference with respect to the Ivy-Bridge processor.

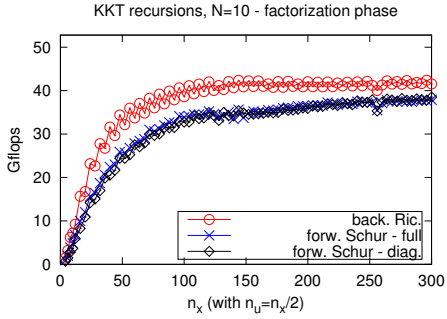
In the factorization routines, HPMPC and MKL roughly double performance with respect to the Ivy-Bridge processor, and therefore the performance ration



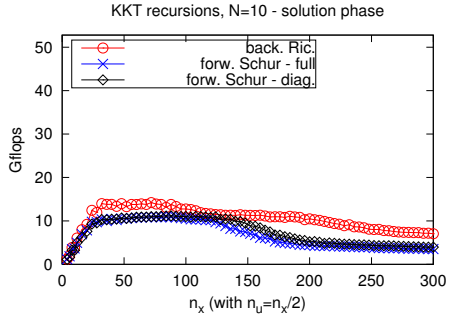
(a) Factorization time.



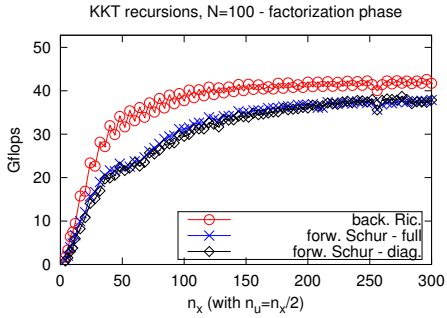
(b) Solution time.



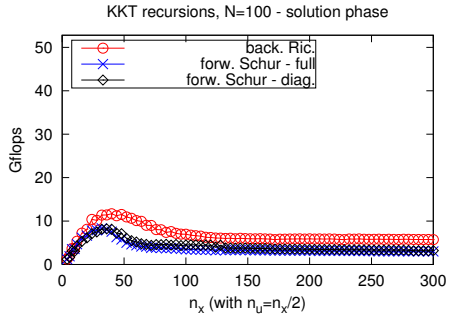
(c) Factorization Gflops.



(d) Solution Gflops.



(e) Factorization Gflops.



(f) Solution Gflops.

Figure 8.4: KKT recursive factorization tests on Intel i7 4800MQ (Intel Haswell micro-architecture, supporting the AVX2 and FMA ISAs).

between the two is about the same. OpenBLAS however shoes a lower performance, especially in the case of the forward Schur-complement recursion in Figure 8.6c, due to the worse state of the AVX2 and FMA ISAs support. Also the reference BLAS and LAPACK versions give a lower performance than in the Ivy-Bridge case, arriving about 10-12% of full FP throughput, showing once again that it is increasingly hard for generic code to give high performance on modern processors. The reference BLAS and LAPACK versions are about 6-8 times slower than the HPMPC versions.

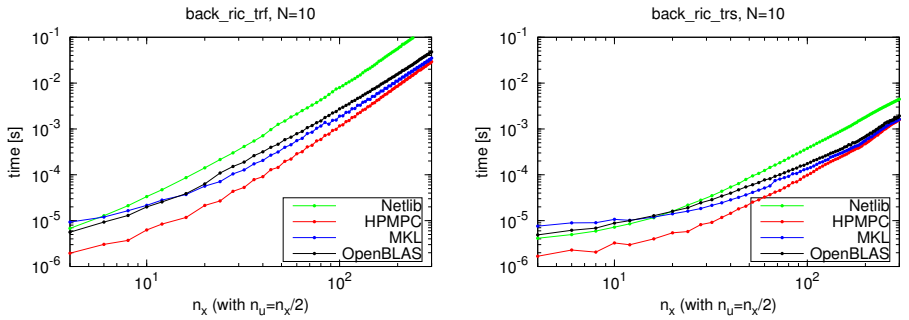
In the solution routines, there is an interesting difference with respect to the Ivy-Bridge ones. Namely, the reference BLAS version gives a much lower performance compared to all the other implementations, also for large-scale problems. This is due to the fact that all reference level 2 BLAS routines where the matrix is transposed perform extremely poorly on the Haswell machine when the FMA ISA is employed (compare e.g. the performance of the 'N' and 'T' variants of the `dgemv` routine in Figures 4.2a and 4.2b). All other versions perform similarly to the Ivy-Bridge case, giving a similar absolute performance (since L3 cache and memory bandwidth are similar), and therefore a lower fraction of the peak FP throughput (since it doubled in the Haswell micro-architecture with respect to the Ivy-Bridge micro-architecture).

8.4 Conclusion

This section presented two structure-exploiting recursive factorizations of the KKT matrix of the unconstrained (linear) MPC and MHE problems. The backward Riccati recursion begins the factorization at the last stage, and it can naturally handle unconstrained MPC problems, but it can not handle additional equality constraints at the last stage. The forward Schur-complement recursion begins the factorization at the first stage, and it can naturally handle unconstrained MHE problems (while it may requires regularization to handle unconstrained MPC problems), and furthermore it can explicitly handle additional equality constraints at the last stage.

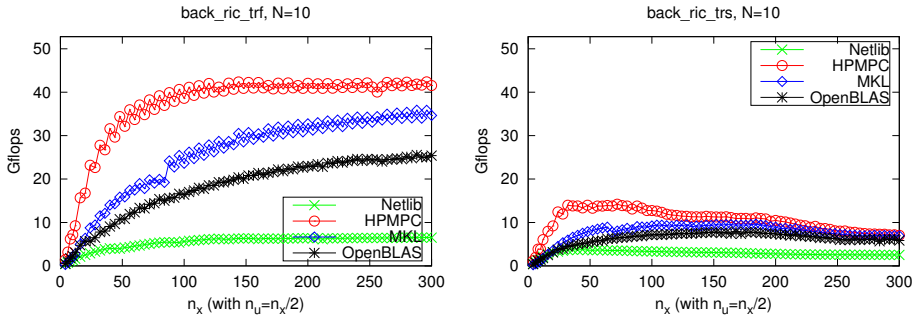
The derivation and implementation (for both standard BLAS and LAPACK routines, and tailored routines in HPMPC) is presented in details. Furthermore, many tests are performed to deeply investigate the performance of the implemented recursions, and their relation to the employed linear algebra routines.

The main findings of the tests for the recursive KKT matrix factorization routines are:



(a) Backward Riccati recursion factorization time.

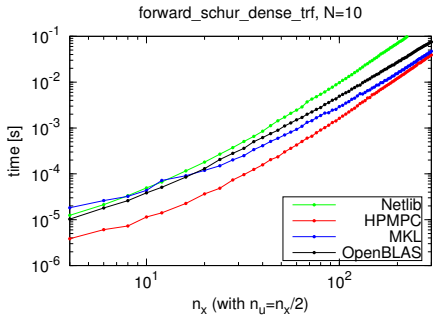
(b) Backward Riccati recursion solution time.



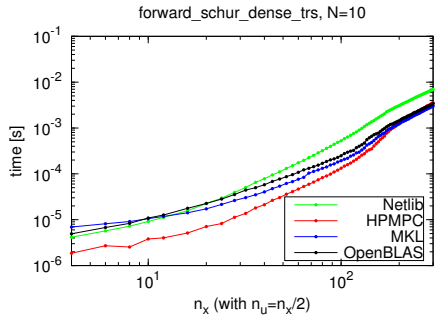
(c) Backward Riccati recursion factorization Gflops.

(d) Backward Riccati recursion solution Gflops.

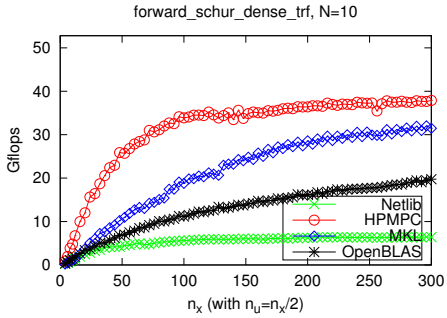
Figure 8.5: Backward Riccati recursion tests on Intel i7 4800MQ (Intel Haswell micro-architecture, supporting the AVX2 and FMA ISAs).



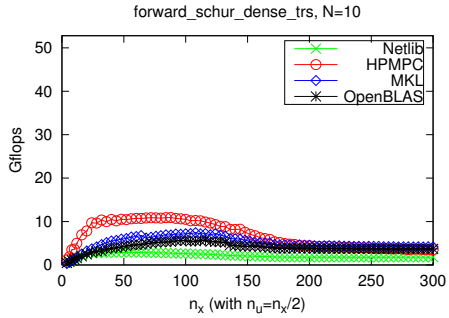
(a) Forward Schur-complement recursion factorization time.



(b) Forward Schur-complement recursion solution time.



(c) Forward Schur-complement recursion factorization Gflops.



(d) Forward Schur-complement recursion solution Gflops.

Figure 8.6: Forward Schur-complement recursion tests on Intel i7 4800MQ (Intel Haswell micro-architecture, supporting the AVX2 and FMA ISAs).

- The main result is that the difference as computation time between different recursive KKT matrix factorization algorithms is smaller than the difference between different implementations.

  The considered factorization routines share the same asymptotic complexity, but with different coefficients. However, the difference in performance for different linear algebra implementations can account for up to an order of magnitude, that is much larger than the ratio between the flop count for the considered factorization routines.

- Regarding the KKT system solution algorithms, the performance is heavily affected by the data memory footprint, as there is no reuse of matrix elements in level 2 BLAS routines, and the performance is bounded by the time to stream data. Difference between implementations is significant only as long as the data memory footprint does not exceed L2 cache size (with the exception of reference BLAS on the Haswell microarchitecture, that gives noticeably worse performance). Therefore, for the small-scale problems typical of embedded optimization, implementation of level 2 BLAS routines can play an important role.

- In case of well-optimized routines, computational throughput is the main factor affecting the performance of the factorization routines, whereas the cache size and the memory bandwidth are the main factors affecting the performance of the solution routines. The latency of FP instructions affects the performance for small-scale problems.

- The backward Riccati recursion is better suited than the forward Schur-complement recursion to be used as a routine in solvers for constrained MPC. In fact, the backward Riccati factorization routine has some performance advantage for small-scale problems, while the backward Riccati solution has some performance advantage for large-scale problems.

- For small-scale problems, the time required to factorize the KKT matrix is only slightly larger than the time required to solve the KKT system once the KKT matrix is factorized. This has important consequences on the choice of optimization algorithms for constrained MPC and MHE problems, since for small-scale problems it would be preferable the choice of optimization algorithms requiring a small amount of combined KKT matrix factorization and solutions (as e.g. in the IPMs), while for large-scale problems it would be preferable the choice of optimization algorithms requiring a small number of KKT matrix factorizations (as e.g. first order methods such as ADMM), since the solution time is about an order of magnitude smaller.

As a final note, the findings of these tests have wider scope than the considered recursive factorization algorithms. In the remainder of the thesis, the tests will

be limited to a single processor, namely the Intel core i7 3520M, supporting the AVX ISA. Furthermore, comparisons with BLAS and LAPACK libraries will not be repeated for other algorithms, the results being analogue.

Chapter 9

Condensing methods

Condensing is traditionally referred to a solution method for the MPC problem, where the states are removed from the problem formulation by using the state-space equation to reformulate them as a function of the initial state (datum) and the inputs (retained as optimization variables). This leads to a smaller but dense and unstructured optimization problem, that can be solved using general-purpose methods: e.g. Cholesky factorization for the unconstrained problem, active set algorithms for the constrained problem.

However, condensing can have a different interpretation: condensing can be seen as a technique that transforms a MPC problem with horizon length N into a MPC problem with horizon length 1, at the expense of increasing the input vector size from n_u to Nn_u. In this interpretation, suggested by the recently proposed work on partial condensing [18], the fact that the states are removed from the problem formulation is just accidental, and due to the fact that the initial state is datum in the MPC problem.

This second interpretation of condensing can naturally be applied to the MHE problem: in fact, in this case the initial state is an optimization variable in both the original and the condensed MHE problems. Therefore the condensed formulation of a MHE problem with horizon N, n_x states and n_u input is just a MHE problem with horizon 1, n_x states and Nn_u inputs.

All numerical tests in this section are performed on a laptop equipped with the Intel Core i7 3520M @ 3.6 GHz in turbo mode, running Linux 14.04. All algorithms are implemented using matrices in the panel-major format, and linear algebra routines part of HPMPC, and described in Part I of this thesis.

As a final note, in the computation of the flop counts, it is assumed that additions and multiplications are performed through a FMA pipeline, and therefore that they have a cost of 2 flops. This reflects the common implementation of these instructions on modern architectures, where generally additions, multiplications and FMA instructions have the same throughput.

9.1 Condensing methods for MPC

Let us assume that $N = 3$. Equation (7.2b) can be rewritten as

$$\bar{x} = \bar{A}^{-1}\bar{B}\bar{u} + \bar{A}^{-1}\bar{b} = \Gamma_u \bar{u} + \Gamma_{x,b} \tag{9.1}$$

where the matrices $\bar{x}$, $\bar{u}$, $\bar{A}$, $\bar{B}$, $\bar{b}$ are

$$\bar{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}, \quad \bar{u} = \begin{bmatrix} u_0 \\ u_1 \\ u_2 \end{bmatrix},$$

$$\bar{A} = \begin{bmatrix} I & & & \\ -A_0 & I & & \\ & -A_1 & I & \\ & & -A_2 & I \end{bmatrix}, \quad \bar{B} = \begin{bmatrix} B_0 & & \\ & B_1 & \\ & & B_2 \end{bmatrix}, \quad \bar{b} = \begin{bmatrix} \hat{x}_0 \\ b_0 \\ b_1 \\ b_2 \end{bmatrix}.$$

Notice that x_0 is considered an optimization variable, and therefore it is part of the $\bar{x}$ vector, even if its value is constrained to $\hat{x}_0$. This choice simplifies the derivation of some algorithms, even if it may be convenient to drop x_0 from the optimization variables to improve the computational time of practical implementations.

Furthermore, notice that the matrix $\bar{A}$ is invertible, sparse (namely block bidiagonal, containing $\mathcal{O}(N)$ non-zeros elements), and furthermore it is lower triangular. The matrix $\bar{A}_N^{-1} = \bar{A}^{-1}$ (where the index N means that the matrix is related to a MPC problem with horizon length N) can be computed recursively by means of the explicit formula for the inversion of a lower triangular matrix

$$\begin{bmatrix} X & \\ Y & Z \end{bmatrix}^{-1} = \begin{bmatrix} X^{-1} & \\ -Z^{-1}YX^{-1} & Z^{-1} \end{bmatrix} \tag{9.2}$$

as

$$\bar{A}_N^{-1} = \begin{bmatrix} \bar{A}_{N-1} & \\ -A_{N-1}\mathcal{E}_{N-1} & I \end{bmatrix}^{-1} = \begin{bmatrix} \bar{A}_{N-1}^{-1} & \\ A_{N-1}\mathcal{E}_{N-1}\bar{A}_{N-1}^{-1} & I \end{bmatrix} \tag{9.3}$$

where $\mathcal{E}_n$ is the matrix

$$\mathcal{E}_n = \begin{bmatrix} 0 & \dots & 0 & I \end{bmatrix} \tag{9.4}$$

of size $n_x \times ((n+1) \cdot n_x)$. Therefore the expression $\mathcal{E}_n \bar{A}_n^{-1}$ is the last block-row of the matrix $\bar{A}_n^{-1}$.

Notice that the matrix $\bar{A}^{-1}$ is dense (namely lower triangular, containing $\mathcal{O}(N^2)$ non-zeros elements), and for $N = 3$ it looks like

$$\bar{A} = \begin{bmatrix} I & & & \\ -A_0 & I & & \\ & -A_1 & I & \\ & & -A_2 & I \end{bmatrix}^{-1} = \begin{bmatrix} I & & & \\ A_0 & I & & \\ A_1A_0 & A_1 & I & \\ A_2A_1A_0 & A_2A_1 & A_2 & I \end{bmatrix} \tag{9.5}$$

Therefore the matrices Γ_u and $\Gamma_{x,b}$ are

$$\Gamma_u = \begin{bmatrix} & & \\ B_0 & & \\ A_1B_0 & B_1 & \\ A_2A_1B_0 & A_2B_1 & B_2 \end{bmatrix}, \quad \Gamma_{x,b} = \begin{bmatrix} \hat{x}_0 \\ A_0\hat{x}_0 + b_0 \\ A_1(A_0\hat{x}_0 + b_0) + b_1 \\ A_2(A_1(A_0\hat{x}_0 + b_0) + b_1) + b_2 \end{bmatrix}.$$

Inserting (9.1) in the cost function

$$\phi = \tfrac{1}{2}\left(\bar{x}^T\bar{Q}\bar{x} + \bar{x}^T\bar{S}^T\bar{u} + \bar{u}^T\bar{S}\bar{x} + \bar{u}^T\bar{R}\bar{u}\right) + \bar{q}^T\bar{x} + \bar{r}^T\bar{u} + \tfrac{1}{2}\rho$$

where the matrices $\bar{Q}$, $\bar{S}$, $\bar{R}$, $\bar{q}$, $\bar{s}$, defined in (7.3), are

$$\bar{Q} = \begin{bmatrix} Q_0 & & & \\ & Q_1 & & \\ & & Q_2 & \\ & & & Q_3 \end{bmatrix}, \quad \bar{S} = \begin{bmatrix} S_0 & & \\ & S_1 & \\ & & S_2 \end{bmatrix}, \quad \bar{R} = \begin{bmatrix} R_0 & & \\ & R_1 & \\ & & R_2 \end{bmatrix},$$

$$\bar{q} = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}, \quad \bar{r} = \begin{bmatrix} r_0 \\ r_1 \\ r_2 \end{bmatrix},$$

the cost function is rewritten as

$$\phi = \tfrac{1}{2}\bar{u}^T H_R \bar{u} + g_r^T \bar{u} + \rho_\rho. \tag{9.6}$$

where

$$H_R = \Gamma_u^T \bar{Q} \Gamma_u + \Gamma_u^T \bar{S}^T + \bar{S} \Gamma_u + \bar{R}$$
$$g_r = \Gamma_u^T \bar{Q} \Gamma_{x,b} + \bar{S} \Gamma_{x,b} + \Gamma_u^T \bar{q} + \bar{r}$$
$$\rho_\rho = \tfrac{1}{2}(\Gamma_{x,b}^T \bar{Q} \Gamma_{x,b} + 2\bar{q}^T \Gamma_{x,b} + \rho)$$

Condensing algorithms can be useful in several cases: to provide the Hessian or the factorized Hessian to gradient and active set methods, as a way to solve unconstrained MPC problems, possibly embedded in interior-point methods. Therefore, in this section several algorithms will be considered: three algorithms to compute the condensed Hessian matrix H_R, two algorithms to compute the factorization of the condensed Hessian matrix, five combined algorithms to compute the Cholesky factor of the Hessian matrix, and finally two algorithms to solve the condensed MPC problem. All these algorithms are characterized by different asymptotic complexity, and therefore are better suited for different problem instances.

9.1.1 Condensing algorithms for MPC

This section contains algorithms to build the condensed Hessian matrix for the MPC problem.

In the first part, efficient algorithms to build Γ_u and $\Gamma_{x,b}$ are presented, that exploit the sparsity of the matrix $\bar{A}$.

Then three different approaches for the computation of the condensed Hessian matrix are presented, and each approach leads to a different asymptotic complexity. The first approach is the classical way to build the condensed Hessian matrix, and it has a computational complexity $\mathcal{O}(N^3)$ and $\mathcal{O}(n_x^2)$. The second approach has been recently proposed in [31] and [16], and it has a computational complexity $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^2)$. The third approach is, to my knowledge, novel, and it employs a recursion that resembles the backward Riccati recursion, giving a complexity $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^3)$, but with a smaller quadratic term in N.

Afterwards, the same three approaches are employed for the computation of the gradient vector. In this case, however, the last two approaches have the same computational complexity.

Finally, the three approaches for the computation of the condensed Hessian matrix are compared, both as number of flops and as execution time of the algorithms implemented using linear algebra routines in HPMPC.

9.1.1.1 $\mathcal{O}(N^2)$ computation of Γ_u

The matrices Γ_u and $\Gamma_{x,b}$ can be efficiently computed in time $\mathcal{O}(N^2)$ and $\mathcal{O}(N)$ respectively. This can be obtained by exploiting the structure of the matrix $\bar{A}$. The key idea is to avoid the explicit computation of the matrix $\bar{A}^{-1}$, and instead directly compute Γ_u and $\Gamma_{x,b}$ using a structure-exploiting lower triangular system solution (i.e. a forward substitution).

For $N = 3$, the generic system to be solved is in the form

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} I & & & \\ -A_0 & I & & \\ & -A_1 & I & \\ & & -A_2 & I \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 - A_0 x_0 \\ x_2 - A_1 x_1 \\ x_3 - A_2 x_2 \end{bmatrix}$$

that gives

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} I & & & \\ -A_0 & I & & \\ & -A_1 & I & \\ & & -A_2 & I \end{bmatrix}^{-1} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 + A_0 x_0 \\ y_2 + A_1 x_1 \\ y_3 + A_2 x_2 \end{bmatrix}. \tag{9.7}$$

Therefore the solution can clearly be computed recursively in time $\mathcal{O}(N)$, as the solution algorithm requires $2Nn_x^2$ flops, where $n_x \times n_x$ is the size of each matrix A_n. In case of a matrix instead of a vector at the right-hand-side, the same algorithm can be applied column-wise.

The computation of $\Gamma_{x,b}$ requires the application of the algorithm to a single column at the right-hand-side, and therefore it has a cost of about $2Nn_x^2$ flops.

The algorithm for the computation of $\Gamma_{x,b}$ is presented in Algorithm 6.

Algorithm 6 Computation of $\Gamma_{x,b}$

1: $\Gamma_{x,b}[0] \leftarrow \hat{x}_0$
2: **for** $i \leftarrow 0, \ldots, N-1$ **do**
3: $\quad \Gamma_{x,b}[i+1] \leftarrow A_i \cdot \Gamma_{x,b}[i] + b_i$
4: **end for**

In the computation of Γ_u, it is possible to exploit the fact that the last elements of each row are 0. The total number of flops can be computed as

$$2n_x^2 n_u \sum_{n=0}^{N-1} n \approx N^2 n_x^2 n_u.$$

The algorithm for the computation of Γ_u is presented in Algorithm 7.

Algorithm 7 Computation of Γ_u

1: $\Gamma_u[1,0] \leftarrow B_0$
2: **for** $i \leftarrow 1, \ldots, N-1$ **do**
3: $\quad \Gamma_u[i+1, 0:i-1] \leftarrow A_i \cdot \Gamma_u[i, 0:i-1]$
4: $\quad \Gamma_u[i+1, i] \leftarrow B_i$
5: **end for**

If the condensing algorithm is implemented using matrices in the panel-major format, then the computation of Γ_u^T is preferred over the computation of Γ_u. In fact, in the computation of Γ_u^T the condensing algorithms operate on block-columns, that are all properly aligned in memory.

The algorithm for the computation of Γ_u^T is presented in Algorithm 8.

Algorithm 8 Computation of Γ_u^T

1: $\Gamma_u^T[0,1] \leftarrow B_0^T$
2: **for** $i \leftarrow 1, \ldots, N-1$ **do**
3: $\quad \Gamma_u^T[0:i-1, i+1] \leftarrow \Gamma_u^T[0:i-1, i] \cdot A_i^T$
4: $\quad \Gamma_u^T[i, i+1] \leftarrow B_i^T$
5: **end for**

Multiplication of matrices on the left by $\bar{A}^{-T}$ can be computed efficiently using a structure-exploiting upper triangular system solution (i.e. a backward substitution). For $N = 3$, the generic system to be solved is in the form

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} I & -A_0^T & & \\ & I & -A_1^T & \\ & & I & -A_2^T \\ & & & I \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} x_0 - A_0^T x_1 \\ x_1 - A_1^T x_2 \\ x_2 - A_2^T x_3 \\ x_3 \end{bmatrix}$$

that gives

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} I & -A_0^T & & \\ & I & -A_1^T & \\ & & I & -A_2^T \\ & & & I \end{bmatrix}^{-1} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} y_0 + A_0^T x_1 \\ y_1 + A_1^T x_2 \\ y_2 + A_2^T x_3 \\ y_3 \end{bmatrix}. \tag{9.8}$$

The cost to compute each column is the same that in the case of the structured forward substitution, and equal to about $2Nn_x^2$ flops.

Multiplication of matrices on the right by $\bar{A}^{-1}$ can be computed efficiently as the transpose of the multiplication of matrices on the left by $\bar{A}^{-T}$, i.e. by computing a row at a time as

$$\begin{bmatrix} x_0^T & x_1^T & x_2^T & x_3^T \end{bmatrix} = \begin{bmatrix} y_0^T + x_1^T A_0 & y_1^T + x_2^T A_1 & y_2^T + x_3^T A_2 & y_3^T \end{bmatrix}. \quad (9.9)$$

9.1.1.2 $\mathcal{O}(N^3)$ and $\mathcal{O}(n_x^2)$ computation of H_R

The key operation in the condensing method is the computation of the matrix $\Gamma_u^T \bar{Q} \Gamma_u$. One natural algorithm is

$$\Gamma_u^T \bar{Q} \Gamma_u = \Gamma_u^T \cdot (\bar{Q} \cdot \Gamma_u)$$

where the Γ_u matrix has been previously computed. If the algorithm is implemented the using the implementation techniques employed in HPMPC, a better algorithm is

$$\Gamma_u^T \bar{Q} \Gamma_u = (\Gamma_u^T \cdot \bar{Q}) \cdot (\Gamma_u^T)^T$$

where the matrix Γ_u^T is precomputed. In this variant, it is possible to operate on block-column sub-matrices (that are properly memory aligned). The matrix-matrix products are computed exploiting the block-triangular structure of the Γ_u matrix.

The matrix $\Gamma_u^T \cdot \bar{Q}$ is

$$\Gamma_u^T \cdot \bar{Q} = \begin{bmatrix} B_0^T Q_1 & B_0^T A_1^T Q_2 & B_0^T A_1^T A_2^T Q_3 \\ & B_1^T Q_2 & B_1^T A_2^T Q_3 \\ & & B_2^T Q_3 \end{bmatrix}$$

and it can be computed one block-column at a time, at a cost of about $N^2 n_x^2 n_u$ flops. Notice that, if the matrices Q_i are diagonal, this cost can be reduced to about $N^2 n_x n_u$ flops, linear in n_x.

Once computed the matrix $\Gamma_u^T \cdot \bar{Q}$, the lower-triangular part of the product $(\Gamma_u^T \cdot \bar{Q}) \cdot \Gamma_u$ is

$$(\Gamma_u^T \cdot \bar{Q}) \cdot \Gamma_u = \begin{bmatrix} T_{00} & * & * \\ T_{10} & T_{11} & * \\ T_{20} & T_{21} & T_{22} \end{bmatrix}$$

where

$$
\begin{aligned}
T_{00} &= B_0^T Q_1 B_0 + B_0^T A_1^T Q_2 A_1 B_0 + B_0^T A_1^T A_2^T Q_3 A_2 A_1 B_0 \\
T_{10} &= B_1^T Q_2 A_1 B_0 + B_1^T A_2^T Q_3 A_2 A_1 B_0 \\
T_{20} &= B_2^T Q_3 A_2 A_1 B_0 \\
T_{11} &= B_1^T Q_2 B_1 + B_1^T A_2^T Q_3 A_2 B_1 \\
T_{21} &= B_2^T Q_3 A_2 B_1 \\
T_{22} &= B_2^T Q_3 B_2
\end{aligned}
$$

and it can be computed using either high- or low-rank updates.

In the case of high-rank updates, the matrix $\Gamma_u^T \bar{Q} \Gamma_u$ is computed one block of size $n_u \times n_u$ at a time. This requires $\frac{N(N-1)}{2}$ calls to the `gemm` BLAS routine and N calls to the `syrk` BLAS routine. The rank of the updates ranges between n_x and Nn_x. If the algorithm is implemented using matrices in panel-major format, this version has the issue that blocks of row index larger than 1 may be not properly aligned in memory. The cost of the algorithm is

$$
2n_x n_u^2 \sum_{m=1}^{N} \sum_{n=1}^{m} n \approx 2n_x n_u^2 \sum_{m=1}^{N} \tfrac{1}{2} m^2 \approx \tfrac{1}{3} N^3 n_x n_u^2
$$

flops, exploiting symmetry.

In the case of low-rank updates, the matrix $\Gamma_u^T \bar{Q} \Gamma_u$ is computed by multiplying the i-th block-column of $\Gamma_u^T \bar{Q}$ by the i-th block-row of Γ_u by means of N calls to the `syrk` BLAS routine. The rank update is fixed to n_x, while the size of the updated sub-matrix ranges from n_u to Nn_u. If the algorithm is implemented using matrices in panel-major format, this version has the advantage that all updated sub-matrices are at the top-left corner, and therefore properly aligned in memory. The cost of the algorithm is the same than in the high-rank update case:

$$
n_x n_u^2 \sum_{n=1}^{N} n^2 \approx \tfrac{1}{3} N^3 n_x n_u^2
$$

flops, exploiting symmetry.

The block-diagonal of the H_R matrix is initialized with the R_i matrices. The strictly block-lower-triangular part of the H_R matrix is initialized with the matrix $\bar{S} \cdot \Gamma_u = (\Gamma_u^T \cdot \bar{S}^T)^T$, that is computed using the `gemm` routine as

$$
\bar{S} \cdot \Gamma_u = \begin{bmatrix} & \\ S_1 B_0 & \\ S_2 A_1 B_0 & S_2 B_1 \end{bmatrix}
$$

at the cost of about $N^2 n_x n_u^2$ flops.

The $\mathcal{O}(N^3)$ condensing algorithm for the low-rank case is summarized in Algorithm 9. Besides the cost to compute Γ_u^T, the cost of the algorithm is of about

$$\tfrac{1}{3}N^3 n_x n_u^2 + N^2 n_x^2 n_u + N^2 n_x n_u^2 \approx \tfrac{1}{3}N^3 n_x n_u^2 + N^2 n_x^2 n_u$$

flops if the matrices Q_i and S_i are full and of about.

$$\tfrac{1}{3}N^3 n_x n_u^2$$

flops if the matrices Q_i are diagonal and the matrices S_i are zero.

Algorithm 9 Computation of H_R, $\mathcal{O}(N^3)$ and $\mathcal{O}(n_x^2)$ algorithm

Require:
Γ_u^T

for $i \leftarrow 0, \dots, N-1$ **do**
  $(\Gamma_u^T \bar{Q})[0:i, i+1] \leftarrow \Gamma_u^T[0:i, i+1] \cdot Q_{i+1}$
end for
for $i \leftarrow 0, \dots, N-1$ **do**
  $H_R[i,i] \leftarrow R_i$
end for
for $i \leftarrow 0, \dots, N-2$ **do**
  $H_R[i+1, 0:i] \leftarrow (\Gamma_u^T[0:i, i+1] \cdot S_{i+1}^T)^T$
end for
for $i \leftarrow 0, \dots, N-1$ **do**
  $H_R[0:i, 0:i] \leftarrow H_R[0:i, 0:i] + (\Gamma_u^T \bar{Q})[0:i, i+1] \cdot (\Gamma_u^T[0:i, i+1])^T$
end for

9.1.1.3 $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^2)$ computation of H_R

By means of the recursive expression for the $\bar{A}^{-1} = \bar{A}_N^{-1}$ matrix in (9.3), it is possible to write the analogue recursive expression for the $\Gamma_u = \Gamma_{u,N}$ matrix

$$\begin{aligned}\Gamma_{u,N} = \bar{A}_N^{-1}\bar{B}_N &= \begin{bmatrix} \bar{A}_{N-1}^{-1} & \\ A_{N-1}\mathcal{E}_{N-1}\bar{A}_{N-1}^{-1} & I \end{bmatrix} \begin{bmatrix} \bar{B}_{N-1} & \\ & B_{N-1} \end{bmatrix} = \\ &= \begin{bmatrix} \bar{A}_{N-1}^{-1}\bar{B}_{N-1} & \\ A_{N-1}\mathcal{E}_{N-1}\bar{A}_{N-1}^{-1}\bar{B}_{N-1} & B_{N-1} \end{bmatrix} = \begin{bmatrix} \Gamma_{u,N-1} & \\ A_{N-1}\mathcal{E}_{N-1}\Gamma_{u,N-1} & B_{N-1} \end{bmatrix}\end{aligned} \tag{9.10}$$

where $\mathcal{E}_n$ is defined in (9.4) and the expression $\mathcal{E}_n\Gamma_{u,n}$ is the last block-row of the matrix $\Gamma_{u,n}$, and similarly the expression $\Gamma_{u,n}^T\mathcal{E}_n^T$ is the last block-column of the matrix $\Gamma_{u,n}^T$.

By means of (9.10), it is possible to investigate the inner structure of the expression $\Gamma_u^T \bar{Q} \Gamma_u = \Gamma_{u,N}^T \bar{Q}_N \Gamma_{u,N}$ as

$$
\begin{aligned}
&(\Gamma_{u,N}^T \bar{Q}_N)\Gamma_{u,N} = \\
&= \begin{bmatrix} \Gamma_{u,N-1}^T & \Gamma_{u,N-1}^T \mathcal{E}_{N-1}^T A_{N-1}^T \\ & B_{N-1}^T \end{bmatrix} \begin{bmatrix} \bar{Q}_{N-1} & \\ & Q_N \end{bmatrix} \begin{bmatrix} \Gamma_{u,N-1} & \\ A_{N-1}\mathcal{E}_{N-1}\Gamma_{u,N-1} & B_{N-1} \end{bmatrix} = \\
&= \begin{bmatrix} \Gamma_{u,N-1}^T \bar{Q}_{N-1} & \Gamma_{u,N-1}^T \mathcal{E}_{N-1}^T A_{N-1}^T Q_N \\ & B_{N-1}^T Q_N \end{bmatrix} \begin{bmatrix} \Gamma_{u,N-1} & \\ A_{N-1}\mathcal{E}_{N-1}\Gamma_{u,N-1} & B_{N-1} \end{bmatrix} = \\
&= \begin{bmatrix} \Gamma_{u,N-1}^T \bar{Q}_{N-1}\Gamma_{u,N-1} + \Gamma_{u,N-1}^T \mathcal{E}_{N-1}^T A_{N-1}^T Q_N A_{N-1}\mathcal{E}_{N-1}\Gamma_{u,N-1} & * \\ (B_{N-1}^T Q_N A_{N-1})\mathcal{E}_{N-1}\Gamma_{u,N-1} & B_{N-1}^T Q_N B_{N-1} \end{bmatrix}
\end{aligned}
$$

Defined $\widetilde{D}_{N-1} = B_{N-1}^T Q_N B_{N-1}$ and $\widetilde{M}_{N-1} = B_{N-1}^T Q_N A_{N-1}$, the last block-row of the matrix $\Gamma_{u,N}^T \bar{Q}_N \Gamma_{u,N}$ can be computed at the cost of $2Nn_x n_u^2$ flops as (using $N = 3$ to make notation easier)

$$
\begin{bmatrix} & & \\ & & \\ \widetilde{M}_2 A_1 B_0 & \widetilde{M}_2 B_1 & \widetilde{D}_2 \end{bmatrix}. \tag{9.11}
$$

The top-left block of $\Gamma_{u,N}^T \bar{Q}_N \Gamma_{u,N}$ has the structure

$$
\begin{aligned}
&\Gamma_{u,N-1}^T \bar{Q}_{N-1}\Gamma_{u,N-1} + \Gamma_{u,N-1}^T \mathcal{E}_{N-1}^T A_{N-1}^T Q_N A_{N-1}\mathcal{E}_{N-1}\Gamma_{u,N-1} = \\
&= \left(\Gamma_{u,N-1}^T \bar{Q}_{N-1} + \Gamma_{u,N-1}^T \mathcal{E}_{N-1}^T A_{N-1}^T Q_N A_{N-1}\mathcal{E}_{N-1}\right)\Gamma_{u,N-1} =
\end{aligned} \tag{9.12}
$$

that has the same structure of the matrix $(\Gamma_{u,N}^T \bar{Q}_N)\Gamma_{u,N}$. In fact, the expression for the matrix $\Gamma_{u,N-1}^T \bar{Q}_{N-1}$ is obtained from the expression for the matrix $\Gamma_{u,N}^T \bar{Q}_N$ with $N-1$ in place of N

$$
\Gamma_{u.N-1}^T \bar{Q}_{N-1} = \begin{bmatrix} \Gamma_{u,N-2}^T \bar{Q}_{N-2} & \Gamma_{u,N-2}^T \mathcal{E}_{N-2}^T A_{N-2}^T Q_{N-1} \\ & B_{N-2}^T Q_{N-1} \end{bmatrix}
$$

and the matrix $\Gamma_{u,N-1}^T \mathcal{E}_{N-1}^T A_{N-1}^T Q_N A_{N-1}\mathcal{E}_{N-1}$ is in the form

$$
\begin{aligned}
&\Gamma_{u,N-1}^T \mathcal{E}_{N-1}^T A_{N-1}^T Q_N A_{N-1}\mathcal{E}_{N-1} = \\
&= \begin{bmatrix} \Gamma_{u,N-2}^T & \Gamma_{u,N-2}^T \mathcal{E}_{N-2}^T A_{N-2}^T \\ & B_{N-2}^T \end{bmatrix} \begin{bmatrix} 0 \\ I \end{bmatrix} A_{N-1}^T Q_N A_{N-1} \begin{bmatrix} 0 & I \end{bmatrix} = \\
&= \begin{bmatrix} 0 & \Gamma_{u,N-2}^T \mathcal{E}_{N-2}^T A_{N-2}^T A_{N-1}^T Q_N A_{N-1} \\ & B_{N-2}^T A_{N-1}^T Q_N A_{N-1} \end{bmatrix}
\end{aligned} \tag{9.13}
$$

where only the last block-column has non-zero elements, and therefore the matrix $\Gamma_{u,N-1}^T \bar{Q}_{N-1} + \Gamma_{u,N-1}^T \mathcal{E}_{N-1}^T A_{N-1}^T Q_N A_{N-1}\mathcal{E}_{N-1}$ can be computed as an

update of the last block-column of the matrix $\Gamma_{u,N-1}^T \bar{Q}_{N-1}$. For $N = 3$, the matrix $\Gamma_{u,N-1}^T \bar{Q}_{N-1} + \Gamma_{u,N-1}^T \mathcal{E}_{N-1}^T A_{N-1}^T Q_N A_{N-1} \mathcal{E}_{N-1}$ is

$$\Gamma_{u,N-1}^T \bar{Q}_{N-1} + \Gamma_{u,N-1}^T \mathcal{E}_{N-1}^T A_{N-1}^T Q_N A_{N-1} \mathcal{E}_{N-1} = \\ = \begin{bmatrix} B_0^T Q_1 & B_0^T A_1^T Q_2 + B_0^T A_1^T A_2^T Q_3 A_2 \\ & B_1^T Q_2 + B_1^T A_2^T Q_3 A_2 \end{bmatrix}$$

where it can be seen that only the last block-column has been updated. The recursion thus can close.

Notice that, exploiting the recursive form of $\Gamma_{u,N}^T \bar{Q}_N$, it holds

$$\begin{bmatrix} \Gamma_{u,N-2}^T \mathcal{E}_{N-2}^T A_{N-2}^T A_{N-1}^T Q_N \\ B_{N-2}^T A_{N-1}^T Q_N \end{bmatrix} = \Gamma_{u,N-1}^T \mathcal{E}_{N-1}^T A_{N-1}^T Q_N$$

that are the first $N-1$ blocks of the last block-column of the matrix $\Gamma_{u,N}^T \bar{Q}_N$. Therefore the matrix in (9.13) can be computed at a cost of $2Nn_x^2 n_u$ flops, besides the cost to compute $\Gamma_u^T \bar{Q}$ (equal to about $N^2 n_x^2 n_u$ flops if $\bar{Q}$ is dense, and to about $N^2 n_x n_u$ if $\bar{Q}$ is block-diagonal). Notice that at the following recursion steps, the matrices to be computed have decreasing size $N-1, \dots, 1$, and summing up over all recursion steps, the computational cost is of about $N^2 n_x^2 n_u$ flops (beside the cost to compute $\Gamma_u^T \bar{Q}$). Therefore this update avoids the computation of $\mathcal{O}(n_x^3)$ operations. In Section 9.1.1.4 a different update is proposed, that makes use of $\mathcal{O}(n_x^3)$ operations to decrease the $\mathcal{O}(N^2)$ terms in the computational complexity.

Furthermore, notice that the computation of the matrix $\bar{S}\Gamma_u$ can be embedded in the computation of the matrix $\Gamma_u^T \bar{Q} \Gamma_u$ at no extra cost. In fact, the last block-row of the matrix $\bar{S}\Gamma_u + \Gamma_u^T \bar{Q}\Gamma_u$ can be computed easily by using

$$M_{N-1} = S_{N-1} + \widetilde{M}_{N-1} = S_{N-1} + B_{N-1}^T Q_N A_{N-1} \tag{9.14}$$

in place of $\widetilde{M}_{N-1}$ in (9.11). Similarly, the last block-diagonal element of the matrix $\bar{Q} + \Gamma_u^T \bar{Q} \Gamma_u$ can be computed easily by using

$$D_{N-1} = R_{N-1} + \widetilde{D}_{N-1} = R_{N-1} + B_{N-1}^T Q_N B_{N-1} \tag{9.15}$$

in place of $\widetilde{D}_{N-1}$ in (9.11).

At the end of the algorithm, the (lower triangular part) condensed Hessian matrix H_R shows the structure

$$H_R = \begin{bmatrix} D_0 & * & * \\ M_1 B_0 & D_1 & * \\ M_2 A_1 B_2 & M_2 B_1 & D_2 \end{bmatrix}.$$

where

$$
\begin{aligned}
D_0 &= R_0 + B_0^T Q_1 B_0 + B_0^T A_1^T Q_2 A_1 B_0 + B_0^T A_1^T A_2^T Q_3 A_2 A_1 B_0 \\
D_1 &= R_1 + B_1^T Q_2 B_1 + B_1^T A_2^T Q_3 A_2 B_1 \\
M_1 &= S_1 + B_1^T Q_2 A_1 + B_1^T A_2^T Q_3 A_2 A_1 \\
D_2 &= R_2 + B_2^T Q_3 B_2 \\
M_2 &= S_2 + B_2^T Q_3 S_2
\end{aligned}
$$

This $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^2)$ condensing algorithm is summarized in Algorithm 10. Beside the cost to compute Γ_u^T, the cost of the algorithm is of about

$$2N^2 n_x^2 n_u + N^2 n_x n_u^2$$

flops if the matrices Q_i are dense, and of about

$$N^2 n_x^2 n_u + N^2 n_x n_u^2$$

flops if the matrices Q_i are diagonal.

Algorithm 10 Computation of H_R, $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^2)$ algorithm

Require:
Γ_u^T

1: $\Gamma_w^T[0:N-1,N] \leftarrow \Gamma_u^T[0:N-1,N] \cdot Q_N$
2: **for** $i \leftarrow N-1,\ldots,1$ **do**
3: $\quad \Gamma_w^T[0:i,i] \leftarrow \Gamma_w^T[0:i,i+1] \cdot A_i$
4: $\quad D_i \leftarrow R_i + B_i^T \cdot (\Gamma_w^T[i,i+1])^T$
5: $\quad H_R[i,i] \leftarrow D_i$
6: $\quad M_i \leftarrow S_i + \Gamma_w^T[i,i]$
7: $\quad H_R[i,0:i-1] \leftarrow (\Gamma_u^T[0:i-1,i] \cdot M_i^T)^T$
8: $\quad \Gamma_w^T[0:i-1,i] \leftarrow \Gamma_w^T[0:i-1,i] + \Gamma_u^T[0:i-1,i] \cdot Q_i$
9: **end for**
10: $D_0 \leftarrow R_0 + B_0^T \cdot (\Gamma_w^T[0,1])^T$
11: $H_R[0,0] \leftarrow D_0$

9.1.1.4 $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^3)$ computation of H_R

It is possible to compute the matrix $\Gamma_u^T \bar{Q} \Gamma_u$ by means of a different algorithm, that makes use of $\mathcal{O}(n_x^3)$ operations in order to reduce the $\mathcal{O}(N^2)$ terms in the

computational complexity of the algorithm. Namely, the update in (9.12) is performed as

$$\Gamma_{u,N-1}^T \bar{Q}_{N-1} \Gamma_{u,N-1} + \Gamma_{u,N-1}^T \mathcal{E}_{N-1}^T A_{N-1}^T Q_N A_{N-1} \mathcal{E}_{N-1} \Gamma_{u,N-1} =$$
$$\Gamma_{u,N-1}^T \left(\bar{Q}_{N-1} + \mathcal{E}_{N-1}^T A_{N-1}^T Q_N A_{N-1} \mathcal{E}_{N-1} \right) \Gamma_{u,N-1}.$$

The matrix $\mathcal{E}_{N-1}^T A_{N-1}^T Q_N A_{N-1} \mathcal{E}_{N-1}$ is zero everywhere but in the bottom-right block. Therefore the update reduces to the update of the bottom-right element of $\bar{Q}_{N-1}$, that is

$$P_{N-1} = Q_{N-1} + A_{N-1}^T P_N A_{N-1}. \tag{9.16}$$

where $P_N = Q_N$. This is an operation with computational cost $\mathcal{O}(n_x^3)$, and it can be computed efficiently in $\frac{7}{3} n_x^3$ flops as

$$P_{N-1} = Q_{N-1} + (A_{N-1}^T \mathcal{L}_N)(A_{N-1}^T \mathcal{L}_N)^T$$

where $\mathcal{L}_N$ is the lower triangular Cholesky factor of P_N.

Summing up over the N stages gives a computational complexity of $\frac{7}{3} N n_x^3$, that replaces a term $2N^2 n_x^2 n_u$ in the computational cost of Algorithm 10. Besides the cost to compute Γ_u^T (equal to about $N^2 n_x^2 n_u$ flops), the cost of the algorithm is of about

$$N^2 n_x n_u^2 + \tfrac{7}{3} N n_x^3 + 3N n_x^2 n_u + N n_x n_u^2 \approx N^2 n_x n_u^2 + \tfrac{7}{3} N n_x^3$$

flops. The algorithm is summarized in Algorithm 11.

9.1.1.5 $\mathcal{O}(N^2)$ computation of g_r

Similar arguments apply to the computation of the right hand side g_r. The key operation of the algorithm is the computation of

$$\Gamma_u^T \cdot (\bar{Q} \Gamma_{x,b} + \bar{q}). \tag{9.17}$$

If the matrix Γ_u (assumed to be precomputed) is used, this operation can be done in $N^2 n_x n_u$ flops, that is quadratic in N and linear in n_x. The algorithm for the computation of the right-hand-side g_r is summarized in Algorithm 12. Besides the cost to compute $\Gamma_{x,b}$, the cost of the algorithm is of about

$$N^2 n_x n_u + 2N n_x^2 + 2N n_x n_u i \approx N^2 n_x n_u + 2N n_x^2$$

flops if the matrices Q_i and S_i are dense, and of about

$$N^2 n_x n_u$$

flops if the matrices Q_i are diagonal and the matrices S_i are zero, that is linear in n_x.

Algorithm 11 Computation of H_R, $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^3)$ algorithm

Require:
Γ_u^T

$P_N \leftarrow Q_N$
for $i \leftarrow N-1, \ldots, 1$ **do**
  $\mathcal{L}_{i+1} \leftarrow P_{i+1}^{1/2}$
  $\begin{bmatrix} B_i^T \\ A_i^T \end{bmatrix} \mathcal{L} \leftarrow \begin{bmatrix} B_i^T \\ A_i^T \end{bmatrix} \cdot \mathcal{L}_{i+1}$
  $\begin{bmatrix} D_i & \\ M_i^T & P_i \end{bmatrix} \leftarrow \begin{bmatrix} R_i & \\ S_i^T & Q_i \end{bmatrix} + \left(\begin{bmatrix} B_i^T \\ A_i^T \end{bmatrix} \mathcal{L} \right) \cdot \left(\begin{bmatrix} B_i^T \\ A_i^T \end{bmatrix} \mathcal{L} \right)^T$
  $H_R[i,i] \leftarrow D_i$
  $H_R[i, 0:i-1] \leftarrow (\Gamma_u^T[0:i-1, i] \cdot M_i^T)^T$
end for
$\mathcal{L}_1 \leftarrow P_1^{1/2}$
$B_0^T \mathcal{L} \leftarrow B_0^T \cdot \mathcal{L}_1$
$D_0 \leftarrow R_0 + \left(B_0^T \mathcal{L}\right) \cdot \left(B_0^T \mathcal{L}\right)^T$
$H_R[0,0] \leftarrow D_0$

Algorithm 12 Computation of g_r, $\mathcal{O}(N^2)$ algorithm

Require:
Γ_u^T, $\Gamma_{x,b}$

for $i \leftarrow 0, \ldots, N-1$ **do**
  $g_r[i] \leftarrow r_i + S_i \cdot \Gamma_{x,b}[i]$
end for
for $i \leftarrow 0, \ldots, N$ **do**
  $(\bar{Q}\Gamma_{x,b} + \bar{q})[i] \leftarrow q_i + Q_i \cdot \Gamma_{x,b}[i]$
end for
for $i \leftarrow 0, \ldots, N-1$ **do**
  $g_r[0:i] \leftarrow g_r[0:i] + \Gamma_u^T[0:i, i+1] \cdot (\bar{Q}\Gamma_{x,b} + \bar{q})[i+1]$
end for

9.1.1.6 $\mathcal{O}(N)$ computation of g_r - (1)

If the structure of the matrix $\Gamma_u = \bar{A}^{-1}\bar{B}$ is exploited, it is possible to compute (9.17) as

$$\Gamma_u^T \cdot (\bar{Q}\Gamma_{x,b} + \bar{q}) = \bar{B}^T \cdot (\bar{A}^{-T} \cdot (\bar{Q}\Gamma_{x,b} + \bar{q}))$$

trading off an increase of the computational complexity in n_x with a reduction in N. In fact, the operation $\bar{A}^{-T} \cdot (\bar{Q}\Gamma_{x,b} + \bar{q})$ can be computed in $2Nn_x^2$ flops using (9.8). The multiplication by $\bar{B}$ can then be computed in $2Nn_xn_u$ flops exploiting the fact that it is block-diagonal. The algorithm for the computation of the right-hand-side g_r is summarized in Algorithm 13. Besides the cost to compute $\Gamma_{x,b}$, the cost of the algorithm is of about

$$4Nn_x^2 + 4Nn_xn_u$$

flops if the matrices Q_i and S_i are dense, and of about

$$2Nn_x^2 + 2Nn_xn_u$$

flops if the matrices Q_i are diagonal and the matrices S_i are zero, that is still quadratic in n_x.

Algorithm 13 Computation of g_r, $\mathcal{O}(N)$ algorithm - (1)

Require:
$\Gamma_{x,b}$

for $i \leftarrow 0, \ldots, N-1$ **do**
  $g_r[i] \leftarrow r_i + S_i \cdot \Gamma_{x,b}[i]$
end for
for $i \leftarrow 0, \ldots, N$ **do**
  $(\bar{Q}\Gamma_{x,b} + \bar{q})[i] \leftarrow q_i + Q_i \cdot \Gamma_{x,b}[i]$
end for
$t[N] \leftarrow (\bar{Q}\Gamma_{x,b} + \bar{q})[N]$
for $i \leftarrow N-1, \ldots, 0$ **do**
  $t[i] \leftarrow (\bar{Q}\Gamma_{x,b} + \bar{q})[i] + A_i^T \cdot t[i+1]$
end for
for $i \leftarrow 0, \ldots, N-1$ **do**
  $g_r[i] \leftarrow g_r[i] + B_i^T \cdot t[i+1]$
end for

9.1.1.7 $\mathcal{O}(N)$ computation of g_r - (2)

The approach of the $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^3)$ method to condense the Hessian matrix can be applied to the condensing of the gradient vector. This algorithm employs

the P_i and M_i matrices computed in Algorithm 11, and therefore its use makes sense only in connection to that Hessian condensing algorithm. Furthermore, the two algorithms could be merged into a single one, similarly to the fact that the backward substitution can be merged with the backward Riccati recursion (see Algorithm 1). For $N = 3$, at the end of the algorithm the gradient vector looks like

$$g_r = \begin{bmatrix} m_0 + M_0\hat{x}_0 \\ m_1 + M_1(A_0\hat{x}_0 + b_0) \\ m_2 + M_2(A_1(A_0\hat{x}_0 + b_0) + b_1) \end{bmatrix}$$

where

$$\begin{aligned} m_0 &= r_0 + B_0^T(P_1 b_0 + P_1) \\ m_1 &= r_1 + B_1^T(P_2 b_1 + P_2) \\ m_2 &= r_2 + B_2^T(P_3 b_2 + P_3) \end{aligned}$$

where in turn

$$\begin{aligned} p_1 &= q_1 + A_1^T(P_2 b_1 + p_2) \\ p_2 &= q_2 + A_2^T(P_3 b_2 + p_3) \\ p_3 &= q_3 \end{aligned}$$

and the matrices P_i and M_i are defined in the Hessian condensing algorithm.

The algorithm is presented in Algorithm 14, and it has a computational complexity of

$$4Nn_x^2 + 4Nn_x n_u$$

flops, irrespective of the fact that the matrices Q_i are dense or diagonal.

9.1.1.8 Comparison of condensing algorithms for MPC

In this section the three condensing algorithms for the computation of H_R are compared, both in terms of flops and in terms of running times of a practical implementation. The algorithms for the computation of g_r are not compared, since this is generally not the key operation.

In the comparison in therm of flops, two cases are considered:

- dense R_i, S_i and Q_i matrices (dense Hessian of the cost function);

Algorithm 14 Computation of g_r, $\mathcal{O}(N)$ algorithm - (2)

Require:
$\Gamma_{x,b}$, P_i, M_i

1: $p_N \leftarrow q_N$
2: **for** $i \leftarrow N-1, \ldots, 1$ **do**
3: $\quad t_i \leftarrow P_{i+1} \cdot b_i + p_{i+1}$
4: $\quad p_i \leftarrow q_i + A_i^T \cdot t_i$
5: $\quad m_i \leftarrow r_i + B_i^T \cdot t_i$
6: $\quad g_r[i] \leftarrow m_i + M_i \cdot \Gamma_{x,b}[i]$
7: **end for**
8: $t_0 \leftarrow P_1 \cdot b_0 + p_1$
9: $m_0 \leftarrow r_0 + B_0^T \cdot t_0$
10: $g_r[0] \leftarrow m_0 + M_0 \cdot \Gamma_{x,b}[0]$

Table 9.1: Comparison of condensing algorithms in terms of flops. In all cases, additional $N^2 n_x^2 n_u$ flops are needed to compute Γ_u^T.

algorithm	dense cost function	diagonal cost function
Algorithm 9	$\frac{1}{3}N^3 n_x n_u^2 + N^2 n_x^2 n_u$	$\frac{1}{3}N^3 n_x n_u^2$
Algorithm 10	$2N^2 n_x^2 n_u + N^2 n_x n_u^2$	$N^2 n_x^2 n_u + N^2 n_x n_u^2$
Algorithm 11	$N^2 n_x n_u^2 + \frac{7}{3}N n_x^3$	$N^2 n_x n_u^2 + \frac{7}{3}N n_x^3$

- diagonal R_i and Q_i matrices and zero S_i matrix (diagonal Hessian of the cost function).

The number of flops for the three condensing algorithms for these two cases are reported in table 9.1. As a rule of thumb, Algorithm 11 is advantageous for smaller values of the ratio N/n_x, while the Algorithm 9 is advantageous for larger values of the ratio N/n_x. Algorithm 10 has the better asymptotic complexity for all dimensions, but with slightly larger coefficients, so this algorithm should work reasonably well in all cases. Algorithms 9 and 10 have a reduced computational complexity in case of diagonal Hessian of the cost function, while Algorithm 11 has the same computational complexity.

In the comparison in terms of running times, the algorithms are implemented using matrices in panel-major format, and using the linear algebra routines in HPMPC. The results are in Figure 9.1. Numerical tests confirm that in general Algorithm 11 is advantageous for smaller values of the ratio N/n_x, while Algorithm 9 is advantageous for larger values of the ratio N/n_x. Algorithm 10 is always the second best. In case of diagonal Hessian of the cost function (case (2)), it is possible to reduce the computational cost of Algorithms 9 and 10, but

not of Algorithm 11, that therefore is convenient only for larger values of the ratio N/n_x.

9.1.2 Factorization algorithms for MPC

In this section two Hessian factorization algorithms are reviewed. The first algorithm is the classical Cholesky factorization algorithm, that is commonly employed to factorize the positive-definite condensed Hessian matrix. The second algorithm is a structure-exploiting Cholesky factorization of the reverse condensed Hessian matrix $\hat{H}_R$, that is the permutation of the condensed Hessian H_R such that the input vector is

$$\hat{u} = \begin{bmatrix} u_{N-1} \\ \vdots \\ u_0 \end{bmatrix} \tag{9.18}$$

The two algorithms have very different computational complexity, and can be combined (with some limitations) with the Hessian condensing algorithms presented in Section 9.1.1.

9.1.2.1 $\mathcal{O}(N^3)$ Cholesky factorization of H_R

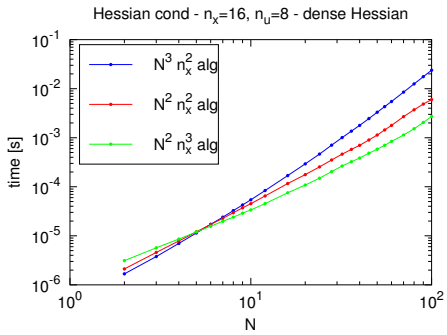
Once built the H_R matrix using the Hessian condensing Algorithms 9, 10 or 11, it is trivially possible to factorize it using the standard Cholesky factorization. More precisely, besides the cost to build the condensed Hessian H_R using Algorithm 9, 10 or 11, the cost of the algorithm is of about

$$\tfrac{1}{3}N^3 n_u^3$$

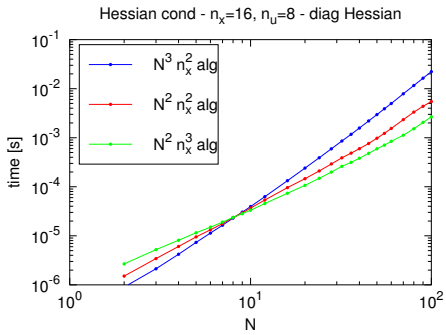
flops, and therefore cubic in both N and n_u, and not dependent on n_x. This is the classical way to factorize H_R.

9.1.2.2 $\mathcal{O}(N)$ Cholesky factorization of $\hat{H}_R$

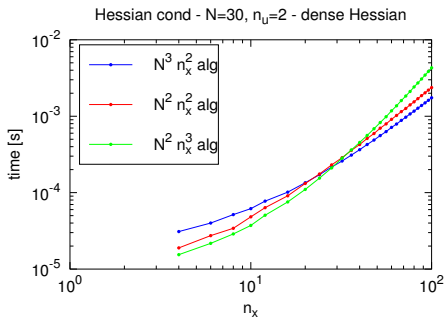
As shown in the paper [34], it is possible to exploit the structure still present in the reversed condensed Hessian matrix $\hat{H}_R$ to compute a structure-exploiting Cholesky factorization that has a computational complexity of $\mathcal{O}(N)$ flops instead of the computational complexity of $\mathcal{O}(N^3)$ flops of the classical Cholesky



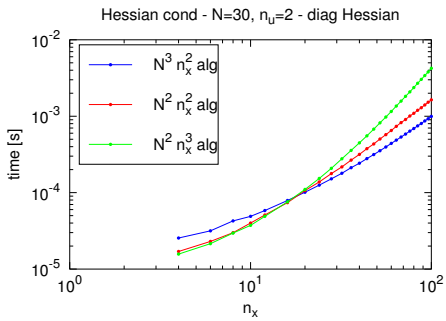
(a) N varying, dense cost function.



(b) N varying, diagonal cost function.



(c) n_x varying, dense cost function.



(d) n_x varying, diagonal cost function.

Figure 9.1: Comparison of Hessian condensing algorithms for MPC: Algorithm 9 with cost $\mathcal{O}(N^3)$ and $\mathcal{O}(n_x^2)$ (blue), Algorithm 10 with cost $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^2)$ (red), Algorithm 11 with cost $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^3)$ (green). The time to compute Γ_u^T is included in the total solution time.

factorization of the condensed Hessian matrix. As a drawback, the computational complexity on n_x is increased, but asymptotically the cost of the factorization algorithm is completely hidden by the cost of the Hessian condensing algorithm.

The main difference between this algorithm and the use of the classical Cholesky factorization of the condensed Hessian H_R is the fact that the condensed Hessian matrix is factorized while build, and therefore the H_R matrix in not computed explicitly and it is not available at the end of the algorithm. The key idea is that the correction part of the Cholesky factorization (accounting for the $\mathcal{O}(N^3)$ term) is replaced by a downgrade of the matrices Q_i at a cost of $Nn_x^2n_u$, similarly to what happens in the Riccati recursion. However this structure-exploiting factorization needs to start from the last stage $N-1$. Since the Cholesky factorization operates from the top-left corner, and the condensed Hessian is traditionally written with the matrices of the first stage 0 at the top-left corner and the matrices of the last stage $N-1$ at the bottom-right corner, there are two options to implement this algorithm

- computation of a reverse Cholesky factorization, that gives the lower triangular factor L_R s.t. $L_R^T \cdot L_R = H_R$. This factorization starts from the bottom-right corner, but there are no standard LAPACK routines for it.
- computation of the classical Cholesky factorization of the permutation $\hat{H}_R$ of the condensed Hessian H_R such that the input vector is (9.18). In this way, the factorization can start from the top-left corner using the standard LAPACK routine `dpotrf`.

In the following, the second approach is considered, due to the use of standard linear algebra routines.

The structure-exploiting factorization procedure boils down to the following three steps, that have to be performed at each stage, starting from the last stage $N-1$ down to the first stage 0:

- computation of the (lower triangular) Cholesky factorization of D_n in (9.15) as $\Lambda_n = D_n^{1/2}$.
- computation of the system solution with M_n in (9.14) as right hand side, as $L_n^T = M_n^T \Lambda_n^{-T}$. This step replaces the solution step in the classical Cholesky factorization, that accounts for the $\mathcal{O}(N^2)$ term in the computational complexity.

- downgrade of the Q_n matrix as $Q_n^* = Q_n - L_n^T L_n$, that is used in the following stage to compute D_{n-1} and M_{n-1}. This step replaces the correction step in the classical Cholesky factorization, that accounts for the $\mathcal{O}(N^3)$ term in the computational complexity.

The lower triangular Cholesky factor $\hat{L}_R$ of the matrix $\hat{H}_R$ is then computed as (using $N = 3$ to make notation easier)

$$\hat{L}_R = \begin{bmatrix} \Lambda_2 & & \\ B_1^T L_2^T & \Lambda_1 & \\ B_0^T A_1^T L_2^T & B_0^T L_1^T & \Lambda_0 \end{bmatrix} \tag{9.19}$$

in the same way as the matrix $\hat{H}_R$ is computed as

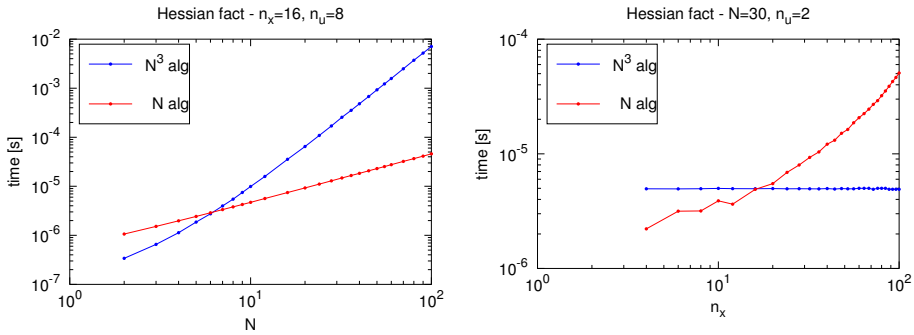
$$\hat{H}_R = \begin{bmatrix} D_2 & & \\ B_1^T M_2^T & D_1 & \\ B_0^T A_1^T M_2^T & B_0^T M_1^T & D_0 \end{bmatrix}.$$

Besides the cost of the Hessian condensing algorithm, the computational complexity of the factorization algorithm is of about

$$N n_x^2 n_u + N n_x n_u^2 + \tfrac{1}{3} N n_u^3$$

flops. When this is added to the computational complexity of the Hessian condensing algorithm, asymptotically the first two terms are totally hidden by the $\mathcal{O}(N^2 n_x^2 n_u)$ and $\mathcal{O}(N^2 n_x n_u^2)$ terms. Therefore, the only term adding to the asymptotic complexity is $\mathcal{O}(N n_u^3)$, that is greatly reduced compared to the $\mathcal{O}(N^3 n_u^3)$ computational complexity of the classical Cholesky factorization of the condensed Hessian.

Since the factorization procedure is embedded in the condensing procedure, a suitable condensing algorithm has to be chosen. Namely, Algorithms 10 and 11 can be used, since they compute explicitly the quantities D_n and M_n. On the other hand, Algorithm 9 can not be used, since it does not provide these quantities, and a modification of Algorithm 9 that explicitly provides D_n and M_n would increase the computational complexity in term of n_x, making the algorithm unattractive with respect to e.g. Algorithm 10. Furthermore notice that, even if Q_n is diagonal, in general Q_n^* is not diagonal for $n \neq N$, and therefore only the versions of the Hessian condensing algorithms considering Q_n as dense can be used.



(a) N varying. **(b)** n_x varying.

Figure 9.2: Comparison of Hessian factorization algorithms for MPC: Cholesky factorization of H_R with cost $\frac{1}{3}N^3n_u^3$ (red), structure exploiting Cholesky factorization of $\hat{H}_R$ with cost $Nn_x^2n_u+Nn_xn_u^2+\frac{1}{3}Nn_u^3$ (blue).

9.1.2.3 Comparison of factorization algorithms for MPC

In Figure 9.2 there is a comparison of the computational complexity of the two Hessian factorization algorithms, besides the computational complexity to build the condensed Hessian matrix using Algorithms 9, 10 or 11. From the pictures it is clear that the classical Cholesky factorization is efficient for small values of the ratio N/n_x, while the structure-exploiting Cholesky factorization is efficient for large values of the ratio N/n_x.

9.1.3 Condensing and factorization algorithms for MPC

In this section the combination of Hessian condensing and factorization algorithms is presented. All Hessian condensing algorithms (possibly tailored to a diagonal Hessian of the cost function) can be combined with the classical $\mathcal{O}(N^3)$ Cholesky factorization. However, only Algorithm 10 and 11 in the version for dense Hessian of the cost function can be combined with the structure-exploiting $\mathcal{O}(N)$ Cholesky factorization. The feasible combinations are summarized in Table 9.2.

In the remainder of the section, the two feasible combinations of Hessian condensing algorithms with the structure-exploiting $\mathcal{O}(N)$ Cholesky factorization algorithm are presented in detail. Finally, different combinations of Hessian

Table 9.2: Feasible combinations of Hessian factorization algorithms and Hessian factorization algorithms in the MPC case.

	condensing algorithms					
	Algorithm 9		Algorithm 10		Algorithm 11	
factorization algorithms	dense Q_n	diag Q_n	dense Q_n	diag Q_n	dense Q_n	diag Q_n
$\mathcal{O}(N^3)$	x	x	x	x	x	x
$\mathcal{O}(N)$			x		x	

condensing and factorization algorithms for the MPC problem are compared.

9.1.3.1 $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^2)$ computation of the Cholesky factor $\hat{L}_R$

The Hessian condensing Algorithm 10 can be easily combined with the $\mathcal{O}(N)$ structure-exploiting reversed Hessian factorization algorithm for the computation of $\hat{L}_R$. The resulting algorithm is summarized in Algorithm 15. Notice that the matrix Γ_u^T internally used by the Algorithm 15 is not permuted, i.e. it is the same as in Algorithm 10: this simplifies the implementation if the matrices are in panel-major format, since the top-left corner of all sub-matrices used in the computation is properly aligned in memory. However, the algorithm builds the lower Cholesky factor $\hat{L}_R$ of the reversed Hessian $\hat{H}_R$: the permutation is performed in line 8 (diagonal bocks) and in the for loop in lines 11-13 (off-diagonal blocks).

Besides the cost to compute Γ_u^T (equal to about $N^2 n_x^2 n_u$ flops), the overall computational complexity of the algorithm is of about

$$2N^2 n_x^2 n_u + N^2 n_x n_u^2 + N n_x^2 n_u + N n_x n_u^2 + \tfrac{1}{3} N n_u^3 \approx 2N^2 n_x^2 n_u + N^2 n_x n_u^2 + \tfrac{1}{3} N n_u^3$$

flops. This means that, asymptotically, the additional cost of Algorithm 15 with respect to Algorithm 10 is equal to $\frac{1}{3} N n_u^3$ (in place of $\frac{1}{3} N^3 n_u^3$ obtained using the classical Cholesky factorization). Notice that, even if Q_i is diagonal, in general Q_i^* is not diagonal for $i \neq N$, and therefore (except for the last stage) it is not possible to reduce the computational cost in case of diagonal Hessian of the cost function.

9.1.3.2 $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^3)$ computation of the Cholesky factor $\hat{L}_R$

Alternatively, the Hessian condensing Algorithm 11 can be combined with the $\mathcal{O}(N)$ structure-exploiting reversed Hessian factorization algorithm for the com-

Algorithm 15 Computation of the lower Cholesky factor $\hat{L}_R$ of $\hat{H}_R$, $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^2)$ algorithm

Require:
Γ_u^T

$Q_N^* \leftarrow Q_N$
$\Gamma_w^T[0:N-1,N] \leftarrow \Gamma_u^T[0:N-1,N] \cdot Q_N^*$
for $i \leftarrow N-1,\ldots,1$ **do**
  $\Gamma_w^T[0:i,i] \leftarrow \Gamma_w^T[0:i,i+1] \cdot A_i$
  $D_i \leftarrow R_i + B_i^T \cdot (\Gamma_w^T[i,i+1])^T$
  $\Lambda_i \leftarrow D_i^{1/2}$
  $\hat{L}_R[N-1-i,N-1-i] \leftarrow \Lambda_i$
  $M_i \leftarrow S_i + \Gamma_w^T[i,i]$
  $L_i^T \leftarrow M_i^T \cdot \Lambda_i^{-T}$
  for $j \leftarrow 0,\ldots,i-1$ **do**
    $\hat{L}_R[N-1-j,N-1-i] \leftarrow \Gamma_u^T[j,i] \cdot L_i^T$
  end for
  $Q_i^* \leftarrow Q_i - L_i^T \cdot L_i$
  $\Gamma_w^T[0:i-1,i] \leftarrow \Gamma_w^T[0:i-1,i] + \Gamma_u^T[0:i-1,i] \cdot Q_i^*$
end for
$D_0 \leftarrow R_0 + B_0^T \cdot (\Gamma_w^T[0,1])^T$
$\Lambda_0 \leftarrow D_0^{1/2}$
$\hat{L}_R[N-1,N-1] \leftarrow \Lambda_0$

putation of $\hat{L}_R$. The resulting algorithm is summarized in Algorithm 16. Notice that by using Q^*_{N-1} in place of Q_{N-1} in equation (9.34), the classical backward Riccati recursion is obtained:

$$P_{N-1} = Q^*_{N-1} + A^T_{N-1} P_N A_{N-1} = Q_{N-1} + A^T_{N-1} P_N A_{N-1} - L^T_{N-1} L_{N-1}.$$

Therefore it is possible to use the classical backward Riccati recursion to compute a structure-exploiting Cholesky factorization of the reversed condensed Hessian $\hat{H}_R$ [19]. It is possible to embed the computation of Λ_n, L_n and Q^*_n with the Cholesky factorization of P_n, in the same way as in the backward Riccati recursion implementation in Algorithm 2: this is done in line 4 of Algorithm 16.

Besides the cost to compute Γ^T_u (equal to about $N^2 n_x^2 n_u$ flops), the computational complexity of the algorithm is of about

$$N^2 n_x n_u^2 + \tfrac{7}{3} N n_x^3 + 4 N n_x^2 n_u + 2 N n_x n_n^2 + \tfrac{1}{3} N n_u^3 \approx N^2 n_x n_u^2 + \tfrac{7}{3} N n_x^3 + \tfrac{1}{3} N n_u^3$$

flops. Notice that the third and the fourth term are excluded from the final approximation of the computational cost, since asymptotically they are totally hidden by the $\mathcal{O}(N^2 n_x^2 n_u)$ and $\mathcal{O}(N^2 n_x n_u^2)$ terms. Also notice that, beside the term $N^2 n_x^2 n_u$ coming from the computation of Γ^T_u and the term $N^2 n_x n_u^2$ coming from the build of the factor $\hat{L}_R$ as in (9.19), the computational complexity is identical to the backward Riccati recursion one, even if computed summing up the complexity of Algorithm 11 and of the $\mathcal{O}(N)$ reversed Hessian factorization algorithm.

9.1.3.3 Comparison of condensing and factorization algorithms for MPC

In Figure 9.3 there is a comparison of algorithms for the computation of the Cholesky factor of the condensed Hessian H_R or of the reversed condensed Hessian $\hat{H}_R$. Namely, three algorithms are compared:

- Algorithm 9 + classical $\mathcal{O}(N^3)$ condensed Hessian Cholesky factorization. This combines the Hessian condensing algorithm and the Hessian factorization algorithm performing better for small values of the ratio N/n_x. The overall algorithm has computational complexity $\mathcal{O}(N^3)$ and $\mathcal{O}(n_x^2)$.
- Algorithm 15. It is a combination of the Hessian condensing algorithm 10 and of the structure-exploiting $\mathcal{O}(N)$ reversed condensed Hessian Cholesky factorization. The overall algorithm has computational complexity $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^2)$.

Algorithm 16 Computation of the lower Cholesky factor $\hat{L}_R$ of $\hat{H}_R$, $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^3)$ algorithm

Require:
Γ_u^T

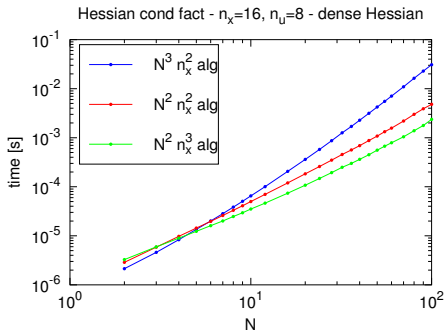
1: $\mathcal{L}_N \leftarrow Q_N^{1/2}$
2: **for** $i \leftarrow N-1, \ldots, 1$ **do**
3: $\quad \begin{bmatrix} B_i^T \\ A_i^T \end{bmatrix} \mathcal{L} \leftarrow \begin{bmatrix} B_i^T \\ A_i^T \end{bmatrix} \cdot \mathcal{L}_{i+1}$
4: $\quad \begin{bmatrix} \Lambda_i & \\ L_i^T & \mathcal{L}_i \end{bmatrix} \leftarrow \left(\begin{bmatrix} R_i & \\ S_i^T & Q_i \end{bmatrix} + \left(\begin{bmatrix} B_i^T \\ A_i^T \end{bmatrix} \mathcal{L} \right) \cdot \left(\begin{bmatrix} B_i^T \\ A_i^T \end{bmatrix} \mathcal{L} \right)^T \right)^{1/2}$
5: $\quad \hat{L}_R[N-1-i, N-1-i] \leftarrow \Lambda_i$
6: $\quad$ **for** $j \leftarrow 0, \ldots, i-1$ **do**
7: $\quad\quad \hat{L}_R[N-1-j, N-1-i] \leftarrow \Gamma_u^T[j,i] \cdot L_i^T$
8: $\quad$ **end for**
9: **end for**
10: $B_0^T \mathcal{L} \leftarrow B_0^T \cdot \mathcal{L}_1$
11: $\Lambda_0 \leftarrow \left(R_0 + \left(B_0^T \mathcal{L} \right) \cdot \left(B_0^T \mathcal{L} \right)^T \right)^{1/2}$
12: $\hat{L}_R[N-1, N-1] \leftarrow \Lambda_0$

- Algorithm 16. It is a combination of the Hessian condensing algorithm 11 and of the structure-exploiting $\mathcal{O}(N)$ reversed condensed Hessian Cholesky factorization, both peforming well for large values of the ratio N/n_x. The overall algorithm has computational complexity $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^3)$.

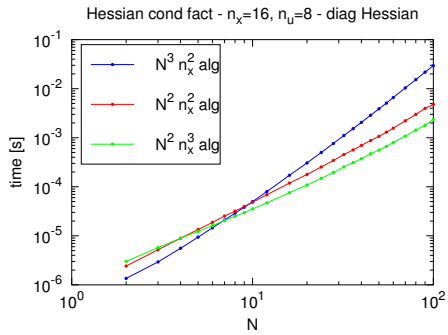
The computational complexity of all other algorithms in Table 9.2 fall above or between the computational complexity of the considered algorithms, so they are not of much interest.

In the comparison in terms of running times, the algorithms are implemented using the matrices in panel-major format, and using the linear algebra routines in HPMPC. The results are in Figure 9.3.

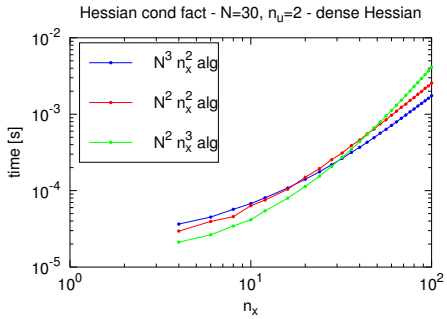
Numerical tests confirms that the combination of the Hessian condensing Algorithm 9 and of the classical $\mathcal{O}(N^3)$ Cholesky factorization performs well for small values of the ratio N/n_x. Furthermore, this combination of algorithms can exploit a diagonal cost function to decrease the computational complexity. On the contrary, the remaining two algorithms can not exploit a diagonal cost function. Algorithm 16 performs well for large values of the ratio N/n_x. Algorithm 15 has a good asymptotic complexity in terms of both N and n_x, and it is the



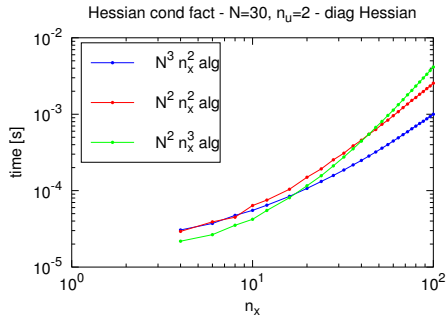
(a) N varying, dense cost function.



(b) N varying, diagonal cost function.



(c) n_x varying, dense cost function.



(d) n_x varying, diagonal cost function.

Figure 9.3: Comparison of Hessian condensing and factorization algorithms for MPC: Algorithm 9 + $\mathcal{O}(N^3)$ Cholesky factorization, with cost $\mathcal{O}(N^3)$ and $\mathcal{O}(n_x^2)$ (blue), Algorithm 15 with cost $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^2)$ (red), Algorithm 16 with cost $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^3)$ (green). The time to compute Γ_u^T is included in the total solution time.

second best in almost all tests.

9.1.4 Solution algorithms for MPC

If the aim of the condensing procedure is the solution of the KKT system of the MPC problem, a solution procedure must be employed after the completion of the KKT matrix factorization procedure.

The classical way to do so is to use the e.g. lower triangular Cholesky factor L_R of the condensed Hessian to perform a forward-backward substitution with g_r as right hand side. The lower Cholesky factor can be computed using one of the presented methods.

Alternatively, the structure still present in the Cholesky factor can be exploited to reduce the computational cost, as proposed in the paper [34].

9.1.4.1 $\mathcal{O}(N^2)$ solution

Once computed the lower Cholesky factor L_R of the condensed Hessian matrix H_R and the condensed gradient g_r, it is possible to compute the minimizer of the cost function (9.6) by setting its gradient to zero, as

$$H_R u + g_r = 0 \quad \Rightarrow \quad u = -L_R^{-T} L_R^{-1} g_r$$

The forward and backward substitutions can be performed using the dense linear algebra routine `trsv` in BLAS, at a total cost of $2N^2 n_u^2$ flops, that is quadratic on both N and n_u and not dependent on n_x.

9.1.4.2 $\mathcal{O}(N)$ solution

If the $\mathcal{O}(N)$ factorization Algorithm is employed to compute the lower triangular Cholesky factor $\hat{L}_R$ of the permuted Hessian matrix $\hat{H}_R$ as in Algorithms 15 or 16, it is possible to exploit the structure still present in $\hat{L}_R$ to reduce the computational complexity in N of the solution algorithm.

Even more, as shown in the paper [34], it is not even necessary to explicitly build the lower Cholesky factor $\hat{L}_R$, reducing the factorization cost by $N^2 n_x n_u^2$ flops (and e.g. making Algorithm 16 linear in N, with a complexity identical

to the backward Riccati recursion). In fact, only the matrices Λ_i and L_i are employed in the solution algorithm.

The lower Cholesky factor $\hat{L}_R$ in (9.19) has the structure

$$\hat{L}_R = \begin{bmatrix} \Lambda_2 & & \\ B_1^T L_2^T & \Lambda_1 & \\ B_0^T A_1^T L_2^T & B_0^T L_1^T & \Lambda_0 \end{bmatrix} = \hat{\Lambda} + \hat{\Gamma}_u^T \hat{L}^T = \hat{\Lambda} + \hat{B}^T \hat{A}^{-T} \hat{L}^T$$

where

$$\hat{\Lambda} = \begin{bmatrix} \Lambda_2 & & \\ & \Lambda_1 & \\ & & \Lambda_0 \end{bmatrix}, \quad \hat{L}^T = \begin{bmatrix} L_2^T & & \\ & L_1^T & \end{bmatrix},$$

$$\hat{B}^T = \begin{bmatrix} B_2^T & & \\ & B_1^T & \\ & & B_0^T \end{bmatrix}, \quad \hat{A}^{-T} = \begin{bmatrix} I & & & \\ -A_2^T & I & & \\ & -A_1^T & I & \\ & & -A_0^T & I \end{bmatrix}^{-1}.$$

By defining the vector $\hat{y} = \hat{L}_R^T \hat{u}$, the forward substitution is in the form

$$\hat{L}_R \hat{y} = \left(\hat{\Lambda} + \hat{B}^T \hat{A}^{-T} \hat{L}^T \right) \hat{y} = -\hat{g}_r$$

that gives y with the recursion

$$\hat{y} = -\hat{\Lambda}^{-1} \left(\hat{g}_r + \hat{B}^T \hat{A}^{-T} \hat{L}^T \hat{y} \right)$$

that for $N = 3$ looks like

$$\begin{bmatrix} y_2 \\ y_1 \\ y_0 \end{bmatrix} = \begin{bmatrix} -\Lambda_2^{-1} (g_2) \\ -\Lambda_1^{-1} \left(g_1 + B_1^T L_2^T y_2 \right) \\ -\Lambda_0^{-1} \left(g_0 + B_0^T A_1^T L_2^T y_2 + B_0^T L_1^T y_1 \right) \end{bmatrix}.$$

Notice that this is a backward recursion with respect to the indexes of the data matrices, due to the permutation of the condensed Hessian.

The backward substitution is in the form

$$\hat{L}_R^T \hat{u} = \left(\hat{\Lambda}^T + \hat{L} \hat{A}^{-1} \hat{B} \right) \hat{u} = \hat{y}$$

that gives the recursion

$$\hat{u} = \hat{\Lambda}^{-T} \left(\hat{y} - \hat{L} \hat{A}^{-1} \hat{B} \hat{u} \right)$$

that for $N = 3$ looks like

$$\begin{bmatrix} u_2 \\ u_1 \\ u_0 \end{bmatrix} = \begin{bmatrix} \Lambda_2^{-T}\left(y_2 - L_2 B_1 u_1 - L_2 A_1 B_0 u_0\right) \\ \Lambda_1^{-T}\left(y_1 - L_1 B_0 u_0\right) \\ \Lambda_0^{-T}\left(y_0\right) \end{bmatrix}.$$

Notice that this is a forward recursion with respect to the indexes of the data matrices, due to the permutation of the condensed Hessian.

The algorithm is summarized in Algorithm 17. The computational cost of the algorithm is of $4Nn_x^2 + 8Nn_x n_u + 2Nn_u^2$ flops, plus enabling a reduction of the cost of the factorization Algorithms 15 and 16 of $N^2 n_x n_u^2$ flops compared with the $\mathcal{O}(N^2)$ solution algorithm.

Algorithm 17 Computation of the solution of the condensed system, $\mathcal{O}(N)$ algorithm

Require:
Λ_i, L_i

$t_{N-1} \leftarrow 0$
$y_{N-1} \leftarrow -\Lambda_{N-1}^{-1}\left(g_{N-1}\right)$
for $i \leftarrow N-2, \ldots, 0$ **do**
  $t_i \leftarrow L_{i+1}^T y_{i+1} + A_{i+1}^T t_{i+1}$
  $y_i \leftarrow -\Lambda_i^{-1}\left(g_i + B_i^T t_i\right)$
end for
$u_0 \leftarrow \Lambda_0^{-T}\left(y_0\right)$
$t_0 \leftarrow B_0 u_0$
for $i \leftarrow 1, \ldots, N-1$ **do**
  $u_i \leftarrow \Lambda_i^{-T}\left(y_i - L_i t_{i-1}\right)$
  $t_i \leftarrow B_i u_i + A_i t_{i-1}$
end for

9.2 Condensing methods for MHE

Assuming that $n_d = 0$ (i.e. that there are no equality constraints on the last stage), equation (7.7b) is

$$\bar{A}\bar{x} = \bar{B}\bar{u} + \bar{b} \tag{9.20}$$

where the matrices $\bar{x}$, $\bar{u}$, $\bar{A}$, $\bar{B}$, $\bar{b}$, defined in (7.8), are (for $N = 3$)

$$\bar{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}, \quad \bar{u} = \begin{bmatrix} u_0 \\ u_1 \\ u_2 \end{bmatrix},$$

$$\bar{A} = \begin{bmatrix} -A_0 & I & & \\ & -A_1 & I & \\ & & -A_2 & I \end{bmatrix}, \quad \bar{B} = \begin{bmatrix} B_0 & & \\ & B_1 & \\ & & B_2 \end{bmatrix}, \quad \bar{b} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}.$$

Notice that also in the MHE case x_0 is considered an optimization variable, and therefore it is part of the $\bar{x}$ vector, but its value is not constrained. Furthermore, notice that in the MHE case the matrix $\bar{A}$ is not invertible, due to the lack of the identity matrix corresponding to the initial state constraint. Invertibility is a key feature in the condensing methods for MPC, and in the MHE case it can be recovered by using different approaches.

Approach 1 Invertibility can be recovered by adding an additional state space equation for a previous stage. Defined the state at stage -1 as $x_{-1} = 0$ and chosen $B_{-1} = I$, the state at stage 0 can be written as

$$\begin{aligned} x_0 &= A_{-1}x_{-1} + B_{-1}u_{-1} + b_{-1} \\ &= A_{-1}0 + Ix_0 + 0 \end{aligned}$$

where the value of the matrix A_{-1} is irrelevant, and the input at stage -1 is equal to the state at stage 0, $u_{-1} = x_0$. By disregarding the state x_{-1} in the state vector, equation (9.20) can be rewritten as

$$\bar{x} = \bar{\mathcal{A}}^{-1}\bar{\mathcal{B}}\bar{v} + \bar{\mathcal{A}}^{-1}\bar{b} \doteq \Gamma_v\bar{v} + \Gamma_b$$

where

$$\bar{\mathcal{A}} = \begin{bmatrix} I & & & \\ -A_0 & I & & \\ & -A_1 & I & \\ & & -A_2 & I \end{bmatrix}, \quad \bar{\mathcal{B}} = \begin{bmatrix} I & & & \\ & B_0 & & \\ & & B_1 & \\ & & & B_2 \end{bmatrix}, \quad \bar{v} = \begin{bmatrix} x_0 \\ u_0 \\ u_1 \\ u_2 \end{bmatrix}$$

and the matrix $\bar{\mathcal{A}}$ is clearly invertible.

Approach 2 Alternatively, the matrix $\bar{A}$ can be directly split such that

$$\bar{A}\bar{x} = \begin{bmatrix} -A_0 & I & & \\ & -A_1 & I & \\ & & -A_2 & I \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} =$$

$$= \begin{bmatrix} I & & & \\ -A_0 & I & & \\ & -A_1 & I & \\ & & -A_2 & I \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} - \begin{bmatrix} I \\ 0 \\ 0 \\ 0 \end{bmatrix} x_0 = \bar{\mathcal{A}}\bar{x} - \hat{\mathcal{E}}_0 x_0.$$

By means of this definition, equation (9.20) can be rewritten as

$$\bar{x} = \bar{\mathcal{A}}^{-1}\hat{\mathcal{E}}_0 x_0 + \bar{\mathcal{A}}^{-1}\bar{B}\bar{u} + \bar{\mathcal{A}}^{-1}\bar{b} = \Gamma_x x_0 + \Gamma_u \bar{u} + \Gamma_b \tag{9.21}$$

$$= \bar{\mathcal{A}}^{-1}\bar{\mathcal{B}}\bar{v} + \bar{\mathcal{A}}^{-1}\bar{b} = \Gamma_v \bar{v} + \Gamma_b \tag{9.22}$$

where

$$\bar{\mathcal{B}} = \begin{bmatrix} \hat{\mathcal{E}}_0 & \bar{B} \end{bmatrix} = \left[\begin{array}{c|ccc} I & B_0 & & \\ & & B_1 & \\ & & & B_2 \end{array}\right], \quad \bar{v} = \begin{bmatrix} x_0 \\ \bar{u} \end{bmatrix} = \left[\begin{array}{c} x_0 \\ \hline u_0 \\ u_1 \\ u_2 \end{array}\right].$$

Equation (9.21) keeps the initial state and the input vectors separated, while equation (9.22) merges them in a single vector, and it is formally identical to (9.1).

The former interpretation is useful when the condensed MHE problem is considered as a MHE problem with horizon length 1 and input size Nn_u, since it allows to clearly identify the different components of the condensed cost function.

The latter interpretation suggests that it is possible to adapt all condensing algorithms developed for the MPC case to the MHE case, provided that the free initial state x_0 is considered as an extra input variable, corresponding to stage -1. Therefore, if the condensing algorithms are coded with the option to have stage-varying number of variables, it is possible to use them in the MHE case straight away. Otherwise, the analogy of equations (9.1) and (9.22) can be exploited to develop condensing algorithms tailored to the MHE case. This will be done in the following, where detailed complexity analysis are performed also for the MHE case.

The matrix $\bar{\mathcal{A}}^{-1} = \bar{\mathcal{A}}_N^{-1}$ (where the index N means that the matrix is related to a MHE problem with horizon length N) can be computed recursively by means of the explicit formula for the inverse of a lower triangular matrix (9.2) as

$$\bar{\mathcal{A}}_N^{-1} = \begin{bmatrix} \bar{\mathcal{A}}_{N-1} & \\ -A_{N-1}\mathcal{E}_{N-1} & I \end{bmatrix}^{-1} = \begin{bmatrix} \bar{\mathcal{A}}_{N-1}^{-1} & \\ A_{N-1}\mathcal{E}_{N-1}\bar{\mathcal{A}}_{N-1}^{-1} & I \end{bmatrix} \tag{9.23}$$

where the $\mathcal{E}_n$ matrix is defined in (9.4). Notice that the matrix $\bar{\mathcal{A}}^{-1}$ is dense (namely lower triangular, containing $\mathcal{O}(N^2)$ non-zeros elements), and for $N = 3$ it looks like

$$\bar{\mathcal{A}}^{-1} = \begin{bmatrix} I & & & \\ -A_0 & I & & \\ & -A_1 & I & \\ & & -A_2 & I \end{bmatrix}^{-1} = \begin{bmatrix} I & & & \\ A_0 & I & & \\ A_1A_0 & A_1 & I & \\ A_2A_1A_0 & A_2A_1 & A_2 & I \end{bmatrix}. \tag{9.24}$$

Therefore the matrices Γ_v and Γ_b are

$$\Gamma_v = \begin{bmatrix} I & & & \\ A_0 & B_0 & & \\ A_1A_0 & A_1B_0 & B_1 & \\ A_2A_1A_0 & A_2A_1B_0 & A_2B_1 & B_2 \end{bmatrix}, \quad \Gamma_b = \begin{bmatrix} b_0 \\ A_1b_0 + b_1 \\ A_2(A_1b_0 + b_1) + b_2 \end{bmatrix}. \tag{9.25}$$

Inserting (9.22) in the cost function expression

$$\phi = \tfrac{1}{2}\left(\bar{x}^T\bar{Q}\bar{x} + \bar{x}^T\bar{S}^T\bar{u} + \bar{u}^T\bar{S}\bar{x} + \bar{u}^T\bar{R}\bar{u}\right) + \bar{q}^T\bar{x} + \bar{r}^T\bar{u} + \tfrac{1}{2}\rho \tag{9.26}$$

where the matrices $\bar{Q}$, $\bar{S}$, $\bar{R}$, $\bar{q}$, $\bar{s}$ are analogue to the ones defined in (7.8), padded with zeros to take into account the fact that x_0 is considered also as input variable,

$$\bar{Q} = \begin{bmatrix} Q_0 & & & \\ & Q_1 & & \\ & & Q_2 & \\ & & & Q_3 \end{bmatrix}, \quad \bar{S} = \begin{bmatrix} S_0 & & & \\ & S_1 & & \\ & & S_2 & \end{bmatrix},$$

$$\bar{R} = \begin{bmatrix} & & & \\ & R_0 & & \\ & & R_1 & \\ & & & R_2 \end{bmatrix}, \quad \bar{q} = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}, \quad \bar{r} = \begin{bmatrix} r_0 \\ r_1 \\ r_2 \end{bmatrix},$$

the cost function is rewritten as

$$\phi = \tfrac{1}{2}\bar{v}^T\mathcal{H}_R\bar{v} + \gamma_r^T\bar{v} + \rho_\rho. \tag{9.27}$$

where

$$\begin{aligned} \mathcal{H}_R &= \Gamma_v^T\bar{Q}\Gamma_v + \Gamma_v^T\bar{S}^T + \bar{S}\Gamma_v + \bar{R} \\ \gamma_r &= \Gamma_v^T\bar{Q}\Gamma_b + \bar{S}\Gamma_b + \Gamma_v^T\bar{q} + \bar{r} \\ \rho_\rho &= \tfrac{1}{2}(\Gamma_b^T\bar{Q}\Gamma_b + 2\bar{q}^T\Gamma_b + \bar{\rho}). \end{aligned}$$

On the other hand, inserting (9.21) in the cost function expression (9.26) where the original definitions in (7.8) for the matrices $\bar{Q}$, $\bar{S}$, $\bar{R}$, $\bar{q}$, $\bar{s}$ are employed, gives

the equivalent formulation

$$\begin{aligned}\phi =& \tfrac{1}{2}\begin{bmatrix} x_0^T & \bar{u}^T \end{bmatrix}\begin{bmatrix} \Gamma_x^T\bar{Q}\Gamma_x & \Gamma_u^T\bar{Q}\Gamma_x + \bar{S}\Gamma_x \\ \Gamma_x^T\bar{Q}\Gamma_u + \Gamma_x^T\bar{S}^T & \Gamma_u^T\bar{Q}\Gamma_u + \Gamma_u^T\bar{S}^T + \bar{S}\Gamma_u + \bar{R} \end{bmatrix}\begin{bmatrix} x_0 \\ \bar{u} \end{bmatrix} + \\ &+ \begin{bmatrix} \Gamma_x^T\bar{Q}\Gamma_b + \Gamma_x^T\bar{q} \\ \Gamma_u^T\bar{Q}\Gamma_b + \bar{S}\Gamma_b + \Gamma_u^T\bar{q} + \bar{r} \end{bmatrix}^T\begin{bmatrix} x_0 \\ \bar{u} \end{bmatrix} + \tfrac{1}{2}\left(\Gamma_b^T\bar{Q}\Gamma_b + 2\bar{q}^T\Gamma_b + \bar{\rho}\right) = \\ &= \tfrac{1}{2}\begin{bmatrix} x_0^T & \bar{u}^T \end{bmatrix}\begin{bmatrix} H_Q & H_S^T \\ H_S & H_R \end{bmatrix}\begin{bmatrix} x_0 \\ \bar{u} \end{bmatrix} + \begin{bmatrix} g_q^T & g_r^T \end{bmatrix}\begin{bmatrix} x_0 \\ \bar{u} \end{bmatrix} + \rho_\rho \end{aligned} \tag{9.28}$$

where the components of the cost function associated with the states or the inputs are clearly recognizable. This gives the equations

$$\mathcal{H}_R = \begin{bmatrix} H_Q & H_S^T \\ H_S & H_R \end{bmatrix}, \quad \gamma_r = \begin{bmatrix} g_q \\ g_r \end{bmatrix}.$$

Notice that the expressions for the Hessian matrix $\mathcal{H}_R$ is formally identical to H_R (and to the equivalent matrix in the MPC case), provided that Γ_v replaces Γ_u. Despite the fact that the initial state x_0 is retained as an optimization variable, the expression for the gradient γ_r is also formally identical to the expression for g_r (and to the equivalent vector in the MPC case). On the other hand, the expressions for H_Q, H_S and g_q are not used in the condensing of the MPC problem, since there the initial state is considered datum and not retained as an optimization variable.

In the following, the algorithms are derived using the formal analogy between $\mathcal{H}_R$ in the MHE problem and H_R in the MPC problem, and matrix partition is employed to emphasize the H_Q, H_S, H_R sub-matrices. The same applies to g_r and to the working matrices.

9.2.1 Condensing algorithms for MHE

9.2.1.1 $\mathcal{O}(N^2)$ computation of Γ_v

Also in the MHE case the matrices Γ_v and Γ_b can be efficiently computed in time $\mathcal{O}(N^2)$ and $\mathcal{O}(N)$ respectively, by exploiting the structure of the matrix $\bar{\mathcal{A}}^{-1}$. Similarly to the MPC case, this can be done by means of (9.7), that in the MHE case also includes the matrix A_0.

The computation of Γ_b requires about $2Nx_x^2$ flops (identical to the MPC case), and the algorithm is presented in Algorithm 18.

Algorithm 18 Computation of Γ_b

$\Gamma_b[0] \leftarrow 0$
for $i \leftarrow 0, \dots, N-1$ **do**
    $\Gamma_b[i+1] \leftarrow A_i \cdot \Gamma_b[i] + b_i$
end for

The computation of Γ_v requires about $N^2 n_x^2 n_u + 2Nn_x^3$ flops (that is $2Nn_x^3$ higher than in the MPC case, and cubic in n_x). The algorithm is presented in Algorithm 19.

Algorithm 19 Computation of Γ_v

$\Gamma_v[0,0] \leftarrow I$
for $i \leftarrow 0, \dots, N-1$ **do**
    $\Gamma_v[i+1, 0:i] \leftarrow A_i \cdot \Gamma_v[i, 0:i]$
    $\Gamma_v[i+1, i+1] \leftarrow B_i$
end for

9.2.1.2 $\mathcal{O}(N^3)$ computation of $\mathcal{H}_R$

This section contains the adaptation to the MHE case of the $\mathcal{O}(N^3)$ and $\mathcal{O}(n_x^2)$ algorithm for the MPC case. However, in the MHE case it is not possible to avoid the $\mathcal{O}(n_x^3)$ terms in the computation cost.

The key operation in the condensing method is the computation of the matrix $\Gamma_v^T \bar{Q} \Gamma_v$. In the MPC case, the algorithm

$$\Gamma_v^T \bar{Q} \Gamma_v = \Gamma_v^T \cdot (\bar{Q} \cdot \Gamma_v)$$

has been employed in order to avoid $\mathcal{O}(n_x^3)$ terms. Since it is not possible to do so in the MHE case, a better algorithm is

$$\Gamma_v^T \bar{Q} \Gamma_v = \Gamma_v^T (\bar{L}_Q \bar{L}_Q^T) \Gamma_v = (\Gamma_v^T \cdot \bar{L}_Q) \cdot (\Gamma_v^T \cdot \bar{L}_Q)^T$$

since it preserves symmetry and reduces the computational cost. The matrix Γ_v is pre-computed. The matrix-matrix products are computed exploiting the block-triangular structure of Γ_v.

The matrix $\bar{L}_Q$ is block-diagonal and it contains the lower Cholesky factors L_i

of the Q_i matrices,

$$\bar{L}_Q = \begin{bmatrix} L_0 & & & \\ & L_1 & & \\ & & L_2 & \\ & & & L_3 \end{bmatrix}$$

and it can be computed at a cost of $\frac{1}{3}Nn_x^3$ flops using the LAPACK routine `potrf`. Notice that, if the matrices Q_i are diagonal, this cost can be reduces to about Nn_x flops, that is linear in n_x.

The matrix $\Gamma_u^T \cdot \bar{L}_Q$ is

$$\Gamma_v^T \cdot \bar{L}_Q = \left[\begin{array}{cccc} L_0 & A_0^T L_1 & A_0^T A_1^T L_2 & A_0^T A_1^T A_2^T L_3 \\ \hline & B_0^T L_1 & B_0^T A_1^T L_2 & B_0^T A_1^T A_2^T L_3 \\ & & B_1^T L_2 & B_1^T A_2^T L_3 \\ & & & B_2^T L_3 \end{array}\right]$$

and it can be computed one block-column at a time, at a cost of about $\frac{1}{2}N^2 n_x^2 n_u + Nn_x^3$ flops, using the BLAS routine `trmm`. Notice that, if the matrices Q_i are diagonal, this cost can be reduced to about $N^2 n_x n_u + 2Nn_x^2$ flops, that is quadratic in n_x.

Once computed the matrix $\Gamma_u^T \cdot \bar{L}_Q$, the lower triangular part of the product $(\Gamma_u^T \cdot \bar{L}_Q) \cdot (\Gamma_u^T \cdot \bar{L}_Q)^T$ is

$$(\Gamma_u^T \cdot \bar{L}_Q) \cdot (\Gamma_u^T \cdot \bar{L}_Q)^T = \left[\begin{array}{c|ccc} T_{-1,-1} & * & * & * \\ \hline T_{0,-1} & T_{0,0} & * & * \\ T_{1,-1} & T_{1,0} & T_{1,1} & * \\ T_{2,-1} & T_{2,0} & T_{2,1} & T_{2,2} \end{array}\right]$$

where

$$\begin{aligned}
T_{-1,-1} &= Q_0 + A_0^T Q_1 A_0 + A_0^T A_1^T Q_2 A_1 A_0 + A_0^T A_1^T A_2^T Q_3 A_2 A_1 A_0 \\
T_{0,-1} &= B_0^T Q_1 A_0 + B_0^T A_1^T Q_2 A_1 A_0 + B_0^T A_1^T A_2^T Q_3 A_2 A_1 A_0 \\
T_{0,0} &= B_0^T Q_1 B_0 + B_0^T A_1^T Q_2 A_1 B_0 + B_0^T A_1^T A_2^T Q_3 A_2 A_1 B_0 \\
T_{1,-1} &= B_1^T Q_2 A_1 A_0 + B_1^T A_2^T Q_3 A_2 A_1 A_0 \\
T_{1,0} &= B_1^T Q_2 A_1 B_0 + B_1^T A_2^T Q_3 A_2 A_1 B_0 \\
T_{1,1} &= B_1^T Q_2 B_1 + B_1^T A_2^T Q_3 A_2 B_1 \\
T_{2,-1} &= B_2^T Q_3 A_2 A_1 A_0 \\
T_{2,0} &= B_2^T Q_3 A_2 A_1 B_0 \\
T_{2,1} &= B_2^T Q_3 A_2 B_1 \\
T_{2,2} &= B_2^T Q_3 B_2
\end{aligned}$$

This operation can be performed using the `syrk` BLAS routine, at a cost of $\frac{1}{3}N^3 n_x n_u^2 + N^2 n_x^2 n_u + N n_x^3$ flops, that is $N^2 n_x^2 n_u + N n_x^3$ higher than in the MPC case.

The the $\mathcal{H}_R$ matrix is initialized with the $\bar{R}$ matrix. The strictly block-lower-triangular part of the $\mathcal{H}_R$ matrix is initialized with the matrix $\bar{S}\cdot\Gamma_v = (\Gamma_v^T \cdot \bar{S}^T)^T$, that is computed using the `gemm` routine as

$$\bar{S}\cdot\Gamma_v = \left[\begin{array}{c|cc} & & \\ \hline S_0 & & \\ S_1A_0 & S_1B_0 & \\ S_2A_1A_0 & S_2A_1B_0 & S_2B_1 \end{array}\right]$$

at the cost of about $N^2 n_x n_u^2 + 2N n_x^2 n_u$ flops.

The $\mathcal{O}(N^3)$ condensing algorithm for the MHE case is summarized in Algorithm 20. Besides the cost to compute Γ_v^T (equal to about $N^2 n_x^2 n_u + 2N n_x^3$ flops), the cost of the algorithm is of about

$$\tfrac{1}{3}N^3 n_x n_u^2 + \tfrac{3}{2}N^2 n_x^2 n_u + \tfrac{7}{3}N n_x^3 + N^2 n_x n_u^2 + 2N n_x^2 n_u \approx \tfrac{1}{3}N^3 n_x n_u^2 + \tfrac{3}{2}N^2 n_x^2 n_u + \tfrac{7}{3}N n_x^3$$

flops if the matrices Q_i and S_i are dense and of about.

$$\tfrac{1}{3}N^3 n_x n_u^2 + N^2 n_x^2 n_u + N n_x^3$$

flops if the matrices Q_i are diagonal and the matrices S_i are zero.

9.2.1.3 $\mathcal{O}(N^2)$ computation of $\mathcal{H}_R$ - (1)

Similarly to the MPC case, by means of the recursive expression for the $\bar{\mathcal{A}}^{-1} = \bar{\mathcal{A}}_N^{-1}$ matrix in (9.23), it is possible to write the analogue recursive expression for the $\Gamma_v = \Gamma_{v,N}$ matrix

$$\begin{aligned}\Gamma_{v,N} = \bar{\mathcal{A}}_N^{-1}\bar{\mathcal{B}}_N &= \begin{bmatrix} \bar{\mathcal{A}}_{N-1}^{-1} & \\ A_{N-1}\mathcal{E}_{N-1}\bar{\mathcal{A}}_{N-1}^{-1} & I \end{bmatrix}\begin{bmatrix} \bar{\mathcal{B}}_{N-1} & \\ & B_{N-1} \end{bmatrix} = \\ &= \begin{bmatrix} \bar{\mathcal{A}}_{N-1}^{-1}\bar{\mathcal{B}}_{N-1} & \\ A_{N-1}\mathcal{E}_{N-1}\bar{\mathcal{A}}_{N-1}^{-1}\bar{\mathcal{B}}_{N-1} & B_{N-1} \end{bmatrix} = \begin{bmatrix} \Gamma_{v,N-1} & \\ A_{N-1}\mathcal{E}_{N-1}\Gamma_{v,N-1} & B_{N-1} \end{bmatrix}\end{aligned} \tag{9.29}$$

where $\mathcal{E}_n$ is defined in (9.4) and the expression $\mathcal{E}_n\Gamma_{v,n}$ is the last block-row of the matrix $\Gamma_{v,n}$, and similarly the expression $\Gamma_{v,n}^T\mathcal{E}_n^T$ is the last block-column of the matrix $\Gamma_{v,n}^T$.

Algorithm 20 Computation of $\mathcal{H}_R$, $\mathcal{O}(N^3)$ algorithm

Require:
Γ_u^T

1: **for** $i \leftarrow 0, \ldots, N$ **do**
2: $L_i \leftarrow Q_i^{1/2}$
3: **end for**
4: **for** $i \leftarrow 0, \ldots, N$ **do**
5: $(\Gamma_v^T \bar{Q})[0:i,i] \leftarrow \Gamma_v^T[0:i,i] \cdot Q_i$
6: **end for**
7: $\mathcal{H}_R[0,0] \leftarrow 0$
8: **for** $i \leftarrow 0, \ldots, N-1$ **do**
9: $\mathcal{H}_R[i+1,i+1] \leftarrow R_i$
10: **end for**
11: **for** $i \leftarrow 0, \ldots, N-1$ **do**
12: $\mathcal{H}_R[i+1,0:i] \leftarrow (\Gamma_v^T[0:i,i] \cdot S_i^T)^T$
13: **end for**
14: **for** $i \leftarrow 0, \ldots, N$ **do**
15: $\mathcal{H}_R[0:i,0:i] \leftarrow \mathcal{H}_R[0:i,0:i] + (\Gamma_v^T \bar{Q})[0:i,i] \cdot (\Gamma_v^T[0:i,i])^T$
16: **end for**

By means of (9.10), it is possible to investigate the inner structure of the expression $\Gamma_v^T \bar{Q} \Gamma_v = \Gamma_{v,N}^T \bar{Q}_N \Gamma_{v,N}$ as

$$
\begin{aligned}
&(\Gamma_{v,N}^T \bar{Q}_N)\Gamma_{v,N} = \\
&= \begin{bmatrix} \Gamma_{v,N-1}^T & \Gamma_{v,N-1}^T \mathcal{E}_{N-1}^T A_{N-1}^T \\ & B_{N-1}^T \end{bmatrix} \begin{bmatrix} \bar{Q}_{N-1} & \\ & Q_N \end{bmatrix} \begin{bmatrix} \Gamma_{v,N-1} & \\ A_{N-1}\mathcal{E}_{N-1}\Gamma_{v,N-1} & B_{N-1} \end{bmatrix} = \\
&= \begin{bmatrix} \Gamma_{v,N-1}^T \bar{Q}_{N-1} & \Gamma_{v,N-1}^T \mathcal{E}_{N-1}^T A_{N-1}^T Q_N \\ & B_{N-1}^T Q_N \end{bmatrix} \begin{bmatrix} \Gamma_{v,N-1} & \\ A_{N-1}\mathcal{E}_{N-1}\Gamma_{v,N-1} & B_{N-1} \end{bmatrix} = \\
&= \begin{bmatrix} \Gamma_{v,N-1}^T \bar{Q}_{N-1}\Gamma_{v,N-1} + \Gamma_{v,N-1}^T \mathcal{E}_{N-1}^T A_{N-1}^T Q_N A_{N-1}\mathcal{E}_{N-1}\Gamma_{v,N-1} & * \\ (B_{N-1}^T Q_N A_{N-1})\mathcal{E}_{N-1}\Gamma_{v,N-1} & B_{N-1}^T Q_N B_{N-1} \end{bmatrix}
\end{aligned}
$$

Defined $\widetilde{D}_{N-1} = B_{N-1}^T Q_N B_{N-1}$ and $\widetilde{M}_{N-1} = B_{N-1}^T Q_N A_{N-1}$, the last block-row of the matrix $\Gamma_{v,N}^T \bar{Q}_N \Gamma_{v,N}$ can be computed at the cost of $2Nn_x n_u^2 + 2n_x n_u^2$ flops as (using $N = 3$ to make notation easier)

$$
\left[\begin{array}{c|ccc} & & & \\ \hline & & & \\ \widetilde{M}_2 A_1 A_0 & \widetilde{M}_2 A_1 B_0 & \widetilde{M}_2 B_1 & \widetilde{D}_2 \end{array}\right]. \tag{9.30}
$$

The top-left block of $\Gamma_{v,N}^T \bar{Q}_N \Gamma_{v,N}$ has the structure

$$\begin{aligned}&\Gamma_{v,N-1}^T \bar{Q}_{N-1} \Gamma_{v,N-1} + \Gamma_{v,N-1}^T \mathcal{E}_{N-1}^T A_{N-1}^T Q_N A_{N-1} \mathcal{E}_{N-1} \Gamma_{v,N-1} = \\ &\left(\Gamma_{v,N-1}^T \bar{Q}_{N-1} + \Gamma_{v,N-1}^T \mathcal{E}_{N-1}^T A_{N-1}^T Q_N A_{N-1} \mathcal{E}_{N-1}\right) \Gamma_{v,N-1}\end{aligned} \tag{9.31}$$

that has the same structure of the matrix $(\Gamma_{v,N}^T \bar{Q}_N)\Gamma_{v,N}$. The proof is analogue to the MPC case. In the MHE case, for $N = 3$, the matrix $\Gamma_{u,N-1}^T \bar{Q}_{N-1} + \Gamma_{u,N-1}^T \mathcal{E}_{N-1}^T A_{N-1}^T Q_N A_{N-1} \mathcal{E}_{N-1}$ is

$$\begin{aligned}&\Gamma_{u,N-1}^T \bar{Q}_{N-1} + \Gamma_{u,N-1}^T \mathcal{E}_{N-1}^T A_{N-1}^T Q_N A_{N-1} \mathcal{E}_{N-1} = \\ &\qquad = \begin{bmatrix} Q_0 & A_0^T Q_1 & A_0^T A_1^T Q_2 + A_0^T A_1^T A_2^T Q_3 A_2 \\ & B_0^T Q_1 & B_0^T A_1^T Q_2 + B_0^T A_1^T A_2^T Q_3 A_2 \\ & & B_1^T Q_2 + B_1^T A_2^T Q_3 A_2 \end{bmatrix}\end{aligned}$$

where again it can be seen that only the last block-column has been updated.

Also in the MHE case the computation of the matrix $\bar{S}\Gamma_v$ can be embedded in the computation of the matrix $\Gamma_v^T \bar{Q} \Gamma_v$ at no extra cost. In fact, the last block-row of the matrix $\bar{S}\Gamma_v + \Gamma_u^T \bar{Q} \Gamma_v$ can be computed easily by using

$$M_{N-1} = S_{N-1} + \widetilde{M}_{N-1} = S_{N-1} + B_{N-1}^T Q_N A_{N-1} \tag{9.32}$$

in place of $\widetilde{M}_{N-1}$ in (9.30). Similarly, the last block-diagonal element of the matrix $\bar{Q} + \Gamma_v^T \bar{Q} \Gamma_v$ can be computed easily by using

$$D_{N-1} = R_{N-1} + \widetilde{D}_{N-1} = R_{N-1} + B_{N-1}^T Q_N B_{N-1} \tag{9.33}$$

in place of $\widetilde{D}_{N-1}$ in (9.30).

At the end of the algorithm, the (lower triangular part) condensed Hessian matrix $\mathcal{H}_R$ shows the structure

$$\mathcal{H}_R = \left[\begin{array}{c|ccc} P_0 & * & * & * \\ \hline M_0 & D_0 & * & * \\ M_1 A_0 & M_1 B_0 & D_1 & * \\ M_2 A_1 A_0 & M_2 A_1 B_0 & M_2 B_1 & D_2 \end{array}\right].$$

where

$$
\begin{aligned}
P_0 &= Q_0 + A_0^T Q_1 A_0 + A_0^T A_1^T Q_2 A_1 A_0 + A_0^T A_1^T A_2^T Q_3 A_2 A_1 A_0 \\
M_0 &= S_0 + B_0^T Q_1 A_0 + B_0^T A_1^T Q_2 A_1 A_0 + B_0^T A_1^T A_2^T Q_3 A_2 A_1 A_0 \\
D_0 &= R_0 + B_0^T Q_1 B_0 + B_0^T A_1^T Q_2 A_1 B_0 + B_0^T A_1^T A_2^T Q_3 A_2 A_1 B_0 \\
D_1 &= R_1 + B_1^T Q_2 B_1 + B_1^T A_2^T Q_3 A_2 B_1 \\
M_1 &= S_1 + B_1^T Q_2 A_1 + B_1^T A_2^T Q_3 A_2 A_1 \\
D_2 &= R_2 + B_2^T Q_3 B_2 \\
M_2 &= S_2 + B_2^T Q_3 S_2
\end{aligned}
$$

This $\mathcal{O}(N^2)$ condensing algorithm is summarized in Algorithm 21. Beside the cost to compute Γ_v^T (equal to about $N^2 n_x^2 n_u + 2N n_x^3$ flops), the cost of the algorithm is of about

$$2N^2 n_x^2 n_u + N^2 n_x n_u^2 + 4N n_x^3$$

flops if the matrices Q_i are dense, and of about

$$N^2 n_x^2 n_u + N^2 n_x n_u^2 + 2N n_x^3$$

flops if the matrices Q_i are diagonal.

Algorithm 21 Computation of $\mathcal{H}_R$, $\mathcal{O}(N^2)$ algorithm - (1)

Require:
Γ_v^T

$\Gamma_w^T[0:N,N] \leftarrow \Gamma_v^T[0:N,N] \cdot Q_N$
for $i \leftarrow N-1,\ldots,0$ **do**
  $\Gamma_w^T[0:i+1,i] \leftarrow \Gamma_w^T[0:i+1,i+1] \cdot A_i$
  $D_i \leftarrow R_i + B_i^T \cdot (\Gamma_w^T[i+1,i+1])^T$
  $\mathcal{H}_R[i+1,i+1] \leftarrow D_i$
  $M_i \leftarrow S_i + \Gamma_w^T[i+1,i]$
  $\mathcal{H}_R[i+1,0:i] \leftarrow (\Gamma_v^T[0:i,i] \cdot M_i^T)^T$
  $\Gamma_w^T[0:i,i] \leftarrow \Gamma_w^T[0:i,i] + \Gamma_v^T[0:i,i] \cdot Q_i$
end for
$P_0 \leftarrow 0 + I^T \cdot (\Gamma_w^T[0,0])^T$
$\mathcal{H}_R[0,0] \leftarrow P_0$

9.2.1.4 $\mathcal{O}(N^2)$ computation of $\mathcal{H}_R$ - (2)

It is possible to compute the matrix $\Gamma_u^T \bar{Q} \Gamma_u$ by means of a different algorithm, that reduces the $\mathcal{O}(N^2)$ terms in the computational complexity of the algorithm.

This algorithm is particularly convenient in the MHE case, since the $\mathcal{O}(Nn_x^3)$ terms are present also in the other condensing algorithms, while this is not the case in the MPC case.

The update in (9.31) is performed as

$$\Gamma_{v,N-1}^T \bar{Q}_{N-1} \Gamma_{v,N-1} + \Gamma_{v,N-1}^T \mathcal{E}_{N-1}^T A_{N-1}^T Q_N A_{N-1} \mathcal{E}_{N-1} \Gamma_{v,N-1} = \Gamma_{v,N-1}^T \left(\bar{Q}_{N-1} + \mathcal{E}_{N-1}^T A_{N-1}^T Q_N A_{N-1} \mathcal{E}_{N-1} \right) \Gamma_{v,N-1}.$$

The matrix $\mathcal{E}_{N-1}^T A_{N-1}^T Q_N A_{N-1} \mathcal{E}_{N-1}$ is zero everywhere but in the bottom-right block. Therefore the update reduces to the update of the bottom-right element of $\bar{Q}_{N-1}$, that is

$$P_{N-1} = Q_{N-1} + A_{N-1}^T P_N A_{N-1}. \tag{9.34}$$

where $P_N = Q_N$. This is an operation with computational cost $\mathcal{O}(n_x^3)$, and it can be computed efficiently in $\frac{7}{3}n_x^3$ flops as

$$P_{N-1} = Q_{N-1} + (A_{N-1}^T \mathcal{L}_N)(A_{N-1}^T \mathcal{L}_N)^T$$

where $\mathcal{L}_N$ is the lower triangular Cholesky factor of P_N.

Summing up over the N stages gives a computational complexity of $\frac{7}{3}Nn_x^3$, that replaces the terms $2N^2n_x^2n_u + 4Nn_x^3$ in the computational cost of Algorithm 21. Besides the cost to compute Γ_v^T (equal to about $N^2n_x^2n_u + 2Nn_x^3$ flops), the cost of the algorithm is of about

$$N^2n_xn_u^2 + \tfrac{7}{3}Nn_x^3 + 5Nn_x^2n_u + Nn_xn_u^2 \approx N^2n_xn_u^2 + \tfrac{7}{3}Nn_x^3$$

flops, where the leading terms are unchanged with respect to the MPC case. The algorithm can not exploit the fact that the matrices Q_i are diagonal to reduce the computational complexity. The algorithm is summarized in Algorithm 22.

9.2.1.5 $\mathcal{O}(N^2)$ computation of γ_r

In the computation of the right hand side γ_r, the key operation of the algorithm is the computation of

$$\Gamma_v^T \cdot (\bar{Q}\Gamma_b + \bar{q}). \tag{9.35}$$

If the matrix Γ_v (assumed to be precomputed) is used, this operation can be done in $N^2n_xn_u + 2Nn_x^2$ flops. The algorithm for the computation of the right-hand-side g_r is summarized in Algorithm 23. Besides the cost to compute Γ_b, the cost of the algorithm is of about

$$N^2n_xn_u + 4Nn_x^2 + 2Nn_xn_u \approx N^2n_xn_u + 4Nn_x^2$$

Algorithm 22 Computation of $\mathcal{H}_R$, $\mathcal{O}(N^2)$ algorithm -(2)

Require:
Γ_v^T

$P_N \leftarrow Q_N$
for $i \leftarrow N-1, \dots, 0$ **do**
  $\mathcal{L}_{i+1} \leftarrow P_{i+1}^{1/2}$
  $\begin{bmatrix} B_i^T \\ A_i^T \end{bmatrix} \mathcal{L} \leftarrow \begin{bmatrix} B_i^T \\ A_i^T \end{bmatrix} \cdot \mathcal{L}_{i+1}$
  $\begin{bmatrix} D_i & \\ M_i^T & P_i \end{bmatrix} \leftarrow \begin{bmatrix} R_i & \\ S_i^T & Q_i \end{bmatrix} + \left(\begin{bmatrix} B_i^T \\ A_i^T \end{bmatrix} \mathcal{L} \right) \cdot \left(\begin{bmatrix} B_i^T \\ A_i^T \end{bmatrix} \mathcal{L} \right)^T$
  $\mathcal{H}_R[i+1, i+1] \leftarrow D_i$
  $\mathcal{H}_R[i+1, 0:i] \leftarrow (\Gamma_v^T[0:i, i] \cdot M_i^T)^T$
end for
$\mathcal{H}_R[0, 0] \leftarrow P_0$

flops if the matrices Q_i are dense, and of about

$$N^2 n_x n_u + 2N n_x^2$$

flops if the matrices Q_i are diagonal.

Algorithm 23 Computation of γ_r, $\mathcal{O}(N^2)$ algorithm

Require:
Γ_v^T, Γ_b

for $i \leftarrow 0, \dots, N$ **do**
  $\gamma_r[i] \leftarrow r_i + S_i \cdot \Gamma_b[i]$
end for
for $i \leftarrow 0, \dots, N$ **do**
  $(\bar{Q}\Gamma_b + \bar{q})[i] \leftarrow q_i + Q_i \cdot \Gamma_b[i]$
end for
for $i \leftarrow 0, \dots, N$ **do**
  $\gamma_r[0:i] \leftarrow \gamma_r[0:i] + \Gamma_v^T[0:i, i] \cdot (\bar{Q}\Gamma_b + \bar{q})[i]$
end for

9.2.1.6 $\mathcal{O}(N)$ computation of γ_r - (1)

If the structure of the matrix $\Gamma_v = \bar{A}^{-1}\bar{B}$ is exploited, it is possible to compute (9.35) as

$$\Gamma_v^T \cdot (\bar{Q}\Gamma_b + \bar{q}) = \bar{B}^T \cdot (\bar{A}^{-T} \cdot (\bar{Q}\Gamma_b + \bar{q}))$$

trading off an increase of the computational complexity in n_x with a reduction in N. In fact, the operation $\bar{A}^{-T} \cdot (\bar{Q}\Gamma_b + \bar{q})$ can be computed in $2Nn_x^2$ flops using (9.8). The multiplication by $\bar{B}$ can then be computed in $2Nn_xn_u$ flops exploiting the fact that it is block-diagonal. The algorithm for the computation of the right-hand-side γ_r is summarized in Algorithm 24. Besides the cost to compute Γ_b, the cost of the algorithm is of about

$$4Nn_x^2 + 4Nn_xn_u$$

flops if the matrices Q_i are dense, and of about

$$2Nn_x^2 + 2Nn_xn_u$$

flops if the matrices Q_i are diagonal.

Algorithm 24 Computation of γ_r, $\mathcal{O}(N)$ algorithm - (1)

Require:
Γ_b

for $i \leftarrow 0, \dots, N$ **do**
  $\gamma_r[i] \leftarrow r_i + S_i \cdot \Gamma_b[i]$
end for
for $i \leftarrow 0, \dots, N$ **do**
  $(\bar{Q}\Gamma_b + \bar{q})[i] \leftarrow q_i + Q_i \cdot \Gamma_b[i]$
end for
$t[N] \leftarrow (\bar{Q}\Gamma_b + \bar{q})[N]$
for $i \leftarrow N-1, \dots, 0$ **do**
  $t[i] \leftarrow (\bar{Q}\Gamma_b + \bar{q})[i] + A_i^T \cdot t[i+1]$
end for
for $i \leftarrow 0, \dots, N$ **do**
  $\gamma_r[i] \leftarrow \gamma_r[i] + B_i^T \cdot t[i]$
end for

9.2.1.7 $\mathcal{O}(N)$ computation of γ_r - (2)

The approach of the $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^3)$ method to condense the Hessian matrix can be applied to the condensing of the gradient vector. This algorithm employs the P_i and M_i matrices computed in Algorithm 11, and therefore its use make sense only in connection to that Hessian condensing algorithm. Furthermore, the two algorithms could be merged into a single one, similarly to the fact that the backward substitution can be merged with the backward Riccati recursion (see Algorithm 1). For $N = 3$, at the end of the algorithm the gradient vector

looks like

$$\gamma_r = \left[\begin{array}{c} p_0 \\ \hline \begin{array}{c} m_0 \\ m_1 + M_1 b_0 \\ m_2 + M_2(A_1 b_0 + b_1) \end{array} \end{array}\right]$$

where

$$\begin{aligned} p_0 &= q_0 + A_0^T(P_1 b_0 + p_1) \\ m_0 &= r_0 + B_0^T(P_1 b_0 + p_1) \\ m_1 &= r_1 + B_1^T(P_2 b_1 + p_2) \\ m_2 &= r_2 + B_2^T(P_3 b_2 + p_3) \end{aligned}$$

where in turn

$$\begin{aligned} p_1 &= q_1 + A_1^T(P_2 b_1 + p_2) \\ p_2 &= q_2 + A_2^T(P_3 b_2 + p_3) \\ p_3 &= q_3 \end{aligned}$$

and the matrices P_i and M_i are defined in the Hessian condensing algorithm.

The algorithm is presented in Algorithm 25, and it has a computational complexity of

$$4Nn_x^2 + 4Nn_x n_u$$

flops, irrespective of the fact that the matrices Q_i are dense or diagonal.

Algorithm 25 Computation of γ_r, $\mathcal{O}(N)$ algorithm - (2)

Require:
Γ_b, P_i, M_i

1: $p_N \leftarrow q_N$
2: **for** $i \leftarrow N-1, \ldots, 0$ **do**
3: $\quad t_i \leftarrow P_{i+1} \cdot b_i + p_{i+1}$
4: $\quad p_i \leftarrow q_i + A_i^T \cdot t_i$
5: $\quad m_i \leftarrow r_i + B_i^T \cdot t_i$
6: $\quad \gamma_r[i+1] \leftarrow m_i + M_i \cdot \Gamma_b[i]$
7: **end for**
8: $\gamma_r[0] \leftarrow p_0$

Table 9.3: Comparison of condensing algorithms in terms of flops. In all cases, additional $N^2 n_x^2 n_u + 2N n_x^3$ flops are needed to compute Γ_v^T.

algorithm	dense cost function	diagonal cost function
Algorithm 20	$\frac{1}{3}N^3 n_x n_u^2 + \frac{3}{2}N^2 n_x^2 n_u + \frac{7}{3}N n_x^3$	$\frac{1}{3}N^3 n_x n_u^2 + N^2 n_x^2 n_u + N n_x^3$
Algorithm 21	$2N^2 n_x^2 n_u + N^2 n_x n_u^2 + 4N n_x^3$	$N^2 n_x^2 n_u + N^2 n_x n_u^2 + 2N n_x^3$
Algorithm 22	$N^2 n_x n_u^2 + \frac{7}{3}N n_x^3$	$N^2 n_x n_u^2 + \frac{7}{3}N n_x^3$

9.2.1.8 Comparison of condensing algorithms for MHE

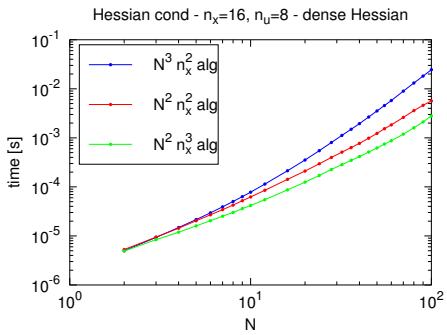
In this section the three condensing algorithm for the computation of $\mathcal{H}_R$ are compared, both in terms of flops and in terms of running times of a practical implementation. The algorithms for the computation of γ_r are not compared, since this is generally not the key operation.

In the comparison in therm of flops, two cases are considered:

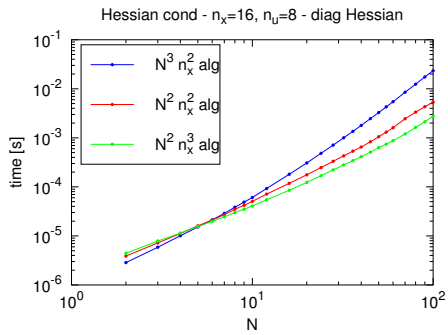
- dense R_i, S_i and Q_i matrices (dense Hessian of the cost function);
- diagonal R_i and Q_i matrices and zero S_i matrix (diagonal Hessian of the cost function).

The number of flops for the three condensing algorithms for these two cases are reported in table 9.3. In the MHE case, all algorithms for the computation of the condensed Hessian matrix $\mathcal{H}_R$ have a computational complexity that is cubic in n_x. Furthermore, Algorithm 22 has the same computational complexity than in the MPC case, and therefore it is likely to be the best choice in most cases. In the case of dense Hessian of the cost function, the computational complexity of Algorihtm 22 is the lowest for all problem sizes. In the case of diagonal Hessian of the cost function, Algorithm 20 could be the best choice if the horizon length is short and the state vector size is large, since it has a smaller coefficient of the $N n_x^3$ term in the computational complexity.

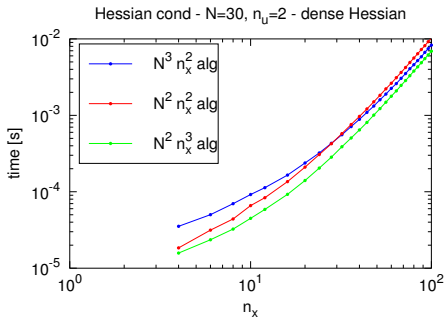
In the comparison in terms of running times, the algorithms are implemented using matrices in panel-major format, and using the linear algebra routines in HPMPC. The results are in Figure 9.4. The numerical results confirm the flop count analysis. In particular, in case of dense Hessian of the cost function, Algorithm 22 is the best choice for all problem sizes. In case of diagonal Hessian of the cost function, Algorithm 22 is the best choice for most problem sizes, with Algorithm 20 that can be slightly faster in case of very short N or very large n_x.



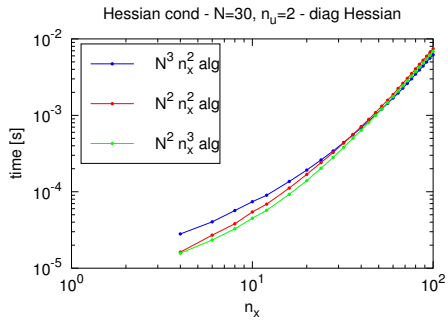
(a) N varying, dense cost function.



(b) N varying, diagonal cost function.



(c) n_x varying, dense cost function.



(d) n_x varying, diagonal cost function.

Figure 9.4: Comparison of Hessian condensing algorithms for MHE: Algorithm 20 (corresponsing to the MPC algorithm 9 with cost $\mathcal{O}(N^3)$ and $\mathcal{O}(n_x^2)$) (blue), Algorithm 21 (corresponsing to the MPC Algorithm 10 with cost $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^2)$) (red), Algorithm 22 (corresponding to the MPC Algorithm 11 with cost $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^3)$) (green). The time to compute Γ_v^T is included in the total solution time.

9.2.2 Factorization algorithms for MHE

In this section two Hessian factorization algorithms are reviewed. The first algorithm is the classical Cholesky factorization algorithm, that is commonly employed to factorize the positive-definite condensed Hessian matrix. The second algorithm is the adaptation to the MHE case of the structure-exploiting Cholesky factorization of the reverse condensed Hessian matrix $\hat{\mathcal{H}}_R$, that is the permutation of the condensed Hessian $\mathcal{H}_R$ such that the input vector is

$$\hat{u} = \left[\begin{array}{c} u_{N-1} \\ \vdots \\ u_0 \\ \hline x_0 \end{array}\right] \tag{9.36}$$

The two algorithms have very different computational complexity, and can be combined (with some limitations) with the Hessian condensing algorithms presented in Section 9.2.1, similarly to the MPC case.

9.2.2.1 $\mathcal{O}(N^3)$ Cholesky factorization of $\mathcal{H}_R$

Once built the $\mathcal{H}_R$ matrix using the Hessian condensing Algorithms 20, 21 or 22, it is trivially possible to factorize it using the standard Cholesky factorization. More precisely, besides the cost to build the condensed Hessian $\mathcal{H}_R$ using Algorithm 20, 21 or 22, in the MHE case the cost of the algorithm is of about

$$\tfrac{1}{3}N^3n_u^3 + N^2n_xn_u^2 + Nn_x^2n_u + \tfrac{1}{3}n_x^3$$

flops, and therefore cubic in $(Nn_u + n_x)$. This is the classical way to factorize $\mathcal{H}_R$.

9.2.2.2 $\mathcal{O}(N)$ Cholesky factorization of $\hat{\mathcal{H}}_R$

Here it is presented the adaptation to the MHE case of the $\mathcal{O}(N)$ Hessian factorization algorithm proposed in the paper [34] for the MPC case.

Since this structure-exploiting factorization starts from the last stage, also in the MHE case it is chosen to compute the classical Cholesky factorization of the permutation $\hat{\mathcal{H}}_R$ of the condensed Hessian $\mathcal{H}_R$ such that the input vector is (9.36).

The only differences with respect to the MPC case is that at the last stage the matrix L_0 needs to be computed, and the matrix P_0 of size $n_x \times n_x$ has to be factorized.

The lower triangular Cholesky factor $\hat{\mathcal{L}}_R$ of the matrix $\hat{\mathcal{H}}_R$ is then computed as (using $N = 3$ to make notation easier)

$$\hat{\mathcal{L}}_R = \left[\begin{array}{ccc|c} \Lambda_2 & & & \\ B_1^T L_2^T & \Lambda_1 & & \\ B_0^T A_1^T L_2^T & B_0^T L_1^T & \Lambda_0 & \\ \hline A_0^T A_1^T L_2^T & A_0^T L_1^T & L_0^T & \mathcal{L}_0 \end{array}\right] \tag{9.37}$$

in the same way as the matrix $\hat{H}_R$ is computed as

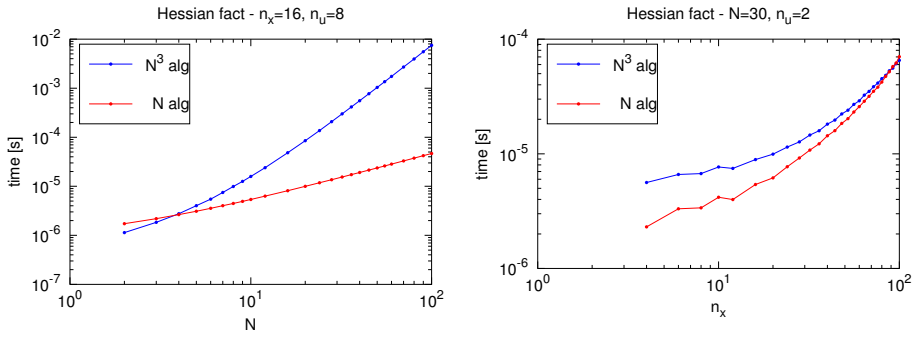
$$\hat{\mathcal{H}}_R = \left[\begin{array}{ccc|c} D_2 & & & \\ B_1^T M_2^T & D_1 & & \\ B_0^T A_1^T M_2^T & B_0^T M_1^T & D_0 & \\ \hline A_0^T A_1^T M_2^T & A_0^T M_1^T & M_0^T & P_0 \end{array}\right].$$

Besides the cost of the Hessian condensing algorithm, the computational complexity of the factorization algorithm is of about

$$\tfrac{1}{3}n_x^3 + Nn_x^2 n_u + Nn_x n_u^2 + \tfrac{1}{3}Nn_u^3$$

flops, that is lower that the classical Cholesky factorization in the MHE case. When this is added to the computational complexity of the Hessian condensing algorithm, asymptotically the first three terms are totally hidden. Therefore, also in the MHE case the only term adding to the asymptotic complexity is $\mathcal{O}(Nn_u^3)$, that is greatly reduced compared to the $\mathcal{O}(N^3 n_u^3)$ computational complexity of the classical Cholesky factorization of the condensed Hessian.

Since the factorization procedure is embedded in the condensing procedure, a suitable condensing algorithm has to be chosen. Namely, Algorithms 21 and 22 can be used, since they compute explicitly the quantities D_n, M_n and P_0. On the other hand, Algorithm 20 can not be used, since it does not provide these quantities, and a modification of Algorithm 20 that explicitly provides D_n, M_n and P_0 would increase the computational complexity, making the algorithm unattractive with respect to e.g. Algorithm 21. Furthermore notice that, even if Q_n is diagonal, in general Q_n^* is not diagonal for $n \neq N$, and therefore only the versions of the Hessian condensing algorithms considering Q_n as dense can be used.



(a) N varying. **(b)** n_x varying.

Figure 9.5: Comparison of Hessian factorization algorithms for MHE: Cholesky factorization of H_R with cost $\frac{1}{3}N^3n_u^3$ (red), structure exploiting Cholesky factorization of $\hat{H}_R$ with cost $Nn_x^2n_u+Nn_xn_u^2+\frac{1}{3}Nn_u^3$ (blue).

9.2.2.3 Comparison of factorization algorithms for MHE

In Figure 9.5 there is a comparison of the computational complexity of the two Hessian factorization algorithms, besides the computational complexity to build the condensed Hessian matrix using Algorithms 20, 21 or 22. In the MHE case, the $\mathcal{O}(N)$ factorization algorithm has a lower computational complexity for all problem sizes. However, from the pictures it appears that the classical Cholesky factorization is slightly more efficient for small values of N, or for very large values of n_x. This is due to the efficiency of implementation: in fact, the Cholesky factorization is a single linear algebra operation that has a large matrix operand, and therefore it has high performance. On the other hand, the $\mathcal{O}(N)$ factorization is implemented as $\mathcal{O}(N)$ calls to linear algebra routines. In particular, beside the $\mathcal{O}(n_x^3)$ factorization of P_0, most of the computational cost comes from the computation of the terms $Q_n^* = Q_n - L_nL_n^T$, that in case of $n_x \gg n_u$ are low-rank updates, and therefore generally attaining a rather low performance. A tailored implementation of the `syrk` routine for the low-rank cases can partially mitigate this performance penalty.

9.2.3 Condensing and factorization algorithms for MHE

In this section the combination of Hessian condensing and factorization algorithms is presented. Similarly to the MPC case, all Hessian condensing algo-

Table 9.4: Feasible combinations of Hessian factorization algorithms and Hessian factorization algorithms in the MHE case.

	condensing algorithms					
	Algorithm 20		Algorithm 21		Algorithm 22	
factorization algorithms	dense Q_n	diag Q_n	dense Q_n	diag Q_n	dense Q_n	diag Q_n
$\mathcal{O}(N^3)$	x	x	x	x	x	x
$\mathcal{O}(N)$			x		x	

rithms (possibly tailored to a diagonal Hessian of the cost function) can be combined with the classical $\mathcal{O}(N^3)$ Cholesky factorization. However, only Algorithm 21 and 22 in the version for dense Hessian of the cost function can be combined with the structure-exploiting $\mathcal{O}(N)$ Cholesky factorization. The feasible combinations are summarized in Table 9.4.

In the remainder of the section, the two feasible combinations of Hessian condensing algorithms with the structure-exploiting $\mathcal{O}(N)$ Cholesky factorization algorithm are presented in detail. Finally, different combinations of Hessian condensing and factorization algorithms for the MHE problem are compared.

9.2.3.1 $\mathcal{O}(N^2)$ computation of the Cholesky factor $\hat{\mathcal{L}}_R$ - (1)

The Hessian condensing Algorithm 21 can be easily combined with the $\mathcal{O}(N)$ structure-exploiting reversed Hessian factorization algorithm for the computation of $\hat{\mathcal{L}}_R$. The resulting algorithm is summarized in Algorithm 26. Notice that the matrix Γ_v^T internally used by the Algorithm 26 is not permuted, i.e. it is the same as in Algorithm 21: this simplifies the implementation if the matrices are in panel-major format, since the top-left corner of all sub-matrices used in the computation is properly aligned in memory. However, the algorithm builds the lower Cholesky factor $\hat{\mathcal{L}}_R$ of the reversed Hessian $\hat{\mathcal{H}}_R$: the permutation is performed in line 7 (diagonal bocks) and in the for loop in lines 10-12 (off-diagonal blocks).

Besides the cost to compute Γ_v^T (equal to about $N^2 n_x^2 n_u + 2N n_x^3$ flops), the overall computational complexity of the algorithm is of about

$$2N^2 n_x^2 n_u + N^2 n_x n_u^2 + 4N n_x^3 + N n_x^2 n_u + N n_x n_u^2 + \tfrac{1}{3} n_x^3 + \tfrac{1}{3} N n_u^3 \approx$$
$$\approx 2N^2 n_x^2 n_u + N^2 n_x n_u^2 + 4N n_x^3 + \tfrac{1}{3} N n_u^3$$

flops. This means that, asymptotically, the additional cost of Algorithm 26 with respect to Algorithm 21 is equal to $\frac{1}{3} N n_u^3$ (in place of $\frac{1}{3}(N n_u + n_x)^3$ obtained

using the classical Cholesky factorization). Notice that, even if Q_i is diagonal, in general Q_i^* is not diagonal for $i \neq N$, and therefore (except for the last stage) it is not possible to reduce the computational cost in case of diagonal Hessian of the cost function.

Algorithm 26 Computation of the lower Cholesky factor $\hat{\mathcal{L}}_R$ of $\hat{\mathcal{H}}_R$, $\mathcal{O}(N^2)$ algorithm - (1)

Require:
Γ_v^T

1: $Q_N^* \leftarrow Q_N$
2: $\Gamma_w^T[0:N,N] \leftarrow \Gamma_v^T[0:N,N] \cdot Q_N$
3: **for** $i \leftarrow N-1,\dots,0$ **do**
4: $\quad \Gamma_w^T[0:i+1,i] \leftarrow \Gamma_w^T[0:i+1,i+1] \cdot A_i$
5: $\quad D_i \leftarrow R_i + B_i^T \cdot (\Gamma_w^T[i+1,i+1])^T$
6: $\quad \Lambda_i \leftarrow D_i^{1/2}$
7: $\quad \hat{\mathcal{L}}_R[N-1-i,N-1-i] \leftarrow \Lambda_i$
8: $\quad M_i \leftarrow S_i + \Gamma_w^T[i+1,i]$
9: $\quad L_i^T \leftarrow M_i^T \cdot \Lambda_i^{-T}$
10: $\quad$ **for** $j \leftarrow 0,\dots,i$ **do**
11: $\quad\quad \hat{\mathcal{L}}_R[N-j,N-1-i] \leftarrow \Gamma_v^T[j,i] \cdot L_i^T$
12: $\quad$ **end for**
13: $\quad Q_i^* \leftarrow Q_i - L_i^T \cdot L_i$
14: $\quad \Gamma_w^T[0:i,i] \leftarrow \Gamma_w^T[0:i,i] + \Gamma_v^T[0:i,i] \cdot Q_i^*$
15: **end for**
16: $P_0 \leftarrow 0 + I^T \cdot (\Gamma_w^T[0,0])^T$
17: $\mathcal{L}_0 \leftarrow P_0^{1/2}$
18: $\hat{\mathcal{L}}_R[N,N] \leftarrow \mathcal{L}_0$

9.2.3.2 $\mathcal{O}(N^2)$ computation of the Cholesky factor $\hat{\mathcal{L}}_R$ - (2)

Alternatively, similarly to the MPC case the Hessian condensing Algorithm 22 can be combined with the $\mathcal{O}(N)$ structure-exploiting reversed Hessian factorization algorithm for the computation of $\hat{\mathcal{L}}_R$. The resulting algorithm is summarized in Algorithm 27. It is possible to embed the computation of Λ_n, L_n and Q_n^* with the Cholesky factorization of P_n, in the same way as in the backward Riccati recursion implementation in Algorithm 2: this is done in line 4 of Algorithm 27.

Besides the cost to compute Γ_u^T (equal to about $N^2 n_x^2 n_u + 2N n_x^3$ flops), the

computational complexity of the algorithm is of about

$$N^2 n_x n_u^2 + \frac{7}{3} N n_x^3 + 6 N n_x^2 n_u + 2 N n_x n_n^2 + \frac{1}{3} n_x^3 + \frac{1}{3} N n_u^3 \approx N^2 n_x n_u^2 + \frac{7}{3} N n_x^3 + \frac{1}{3} N n_u^3$$

flops. Notice that the third, the fourth and the fifth term are excluded from the final approximation of the computational cost, since asymptotically they are totally hidden by the $\mathcal{O}(N^2 n_x^2 n_u)$ and $\mathcal{O}(N^2 n_x n_u^2)$ terms. The leading terms of the computational complexity are identical to the ones in the MPC case.

Algorithm 27 Computation of the lower Cholesky factor $\hat{\mathcal{L}}_R$ of $\hat{\mathcal{H}}_R$, $\mathcal{O}(N^2)$ algorithm - (2)

Require:
Γ_v^T

1: $\mathcal{L}_N \leftarrow Q_N^{1/2}$
2: **for** $i \leftarrow N-1, \dots, 0$ **do**
3: $\begin{bmatrix} B_i^T \\ A_i^T \end{bmatrix} \mathcal{L} \leftarrow \begin{bmatrix} B_i^T \\ A_i^T \end{bmatrix} \cdot \mathcal{L}_{i+1}$
4: $\begin{bmatrix} \Lambda_i & \\ L_i^T & \mathcal{L}_i \end{bmatrix} \leftarrow \left(\begin{bmatrix} R_i & \\ S_i^T & Q_i \end{bmatrix} + \left(\begin{bmatrix} B_i^T \\ A_i^T \end{bmatrix} \mathcal{L} \right) \cdot \left(\begin{bmatrix} B_i^T \\ A_i^T \end{bmatrix} \mathcal{L} \right)^T \right)^{1/2}$
5: $\hat{\mathcal{L}}_R[N-1-i, N-1-i] \leftarrow \Lambda_i$
6: **for** $j \leftarrow 0, \dots, i$ **do**
7: $\hat{\mathcal{L}}_R[N-j, N-1-i] \leftarrow \Gamma_v^T[j,i] \cdot L_i^T$
8: **end for**
9: **end for**
10: $\hat{\mathcal{L}}_R[N,N] \leftarrow \mathcal{L}_0$

9.2.3.3 Comparison of condensing and factorization algorithms for MHE

In Figure 9.6 there is a comparison of algorithms for the computation of the Cholesky factor of the condensed Hessian $\mathcal{H}_R$ or of the reversed condensed Hessian $\hat{\mathcal{H}}_R$. The same three algorithm considered in the MPC case are compared also in the MHE case. Namely, the three algorithms are:

- Algorithm 20 + classical $\mathcal{O}(N^3)$ condensed Hessian Cholesky factorization.
- Algorithm 26. It is a combination of the Hessian condensing algorithm 21 and of the structure-exploiting $\mathcal{O}(N)$ reversed condensed Hessian Cholesky factorization.

- Algorithm 27. It is a combination of the Hessian condensing algorithm 22 and of the structure-exploiting $\mathcal{O}(N)$ reversed condensed Hessian Cholesky factorization.

In the comparison in terms of running times, the algorithms are implemented using the matrices in panel-major format, and using the linear algebra routines in HPMPC. The results are in Figure 9.6. The results are similar to the result of the condensing algorithm for the MHE case in Figure 9.4, since the factorization cost is generally smaller than the condensing cost. Namely, in case of dense Hessian of the cost function, Algorithm 27 is the best choice for all problem sizes. In case of diagonal Hessian of the cost function, Algorithm 27 is the best choice for most problem sizes, with Algorithm 20 + classical $\mathcal{O}(N^3)$ Cholesky factorization being slightly faster in case of very small N or very large n_x, due to the smaller coefficient of the $\mathcal{O}(Nn_x^3)$ term in Algorithm 20.

9.2.4 Solution algorithms for MHE

If the aim of the condensing procedure is the solution of the KKT system of the MPC problem, a solution procedure must be employed after the completion of the KKT matrix factorization procedure.

The classical way to do so is to use the e.g. lower triangular Cholesky factor $\mathcal{L}_R$ of the condensed Hessian to perform a forward-backward substitution with γ_r as right hand side. The lower Cholesky factor can be computed using one of the presented methods.

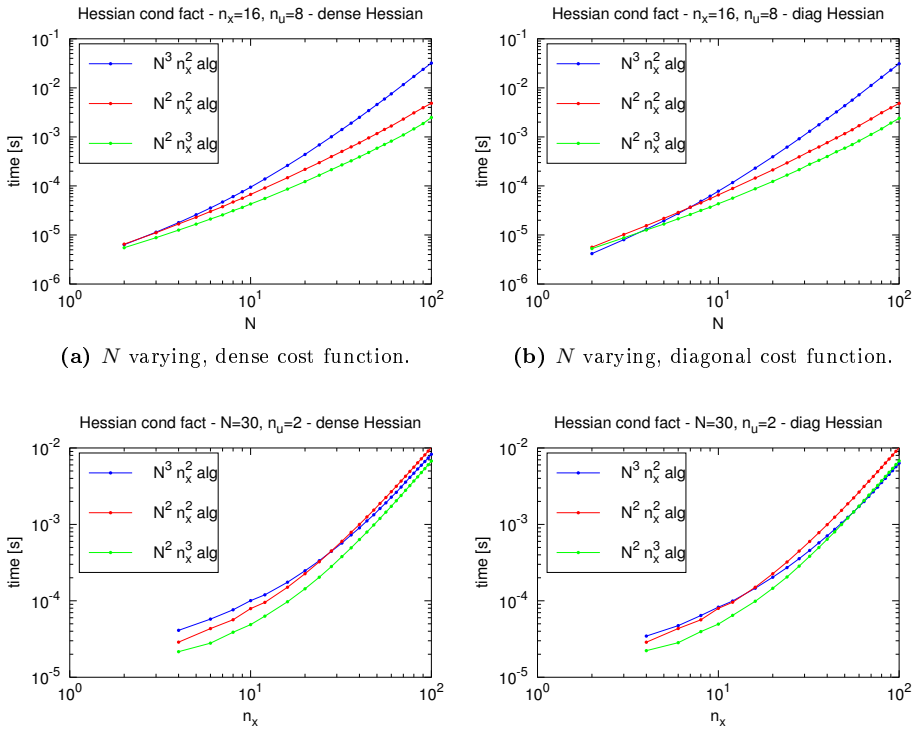
Alternatively, the structure still present in the Cholesky factor can be exploited to reduce the computational cost, as proposed in the paper [34] for the MPC case.

9.2.4.1 $\mathcal{O}(N^2)$ solution

Once computed the lower Cholesky factor $\mathcal{L}_R$ of the condensed Hessian matrix $\mathcal{H}_R$ and the condensed gradient γ_r, it is possible to compute the minimizer of the cost function (9.27) by setting its gradient to zero, as

$$\mathcal{H}_R u + \gamma_r = 0 \quad \Rightarrow \quad u = -\mathcal{L}_R^{-T}\mathcal{L}_R^{-1}\gamma_r$$

The forward and backward substitutions can be performed using the dense linear algebra routine `trsv` in BLAS, at a total cost of $2(Nn_u + n_x)^2$ flops, that is quadratic on Nn_u and n_x.



(a) N varying, dense cost function.

(b) N varying, diagonal cost function.

(c) n_x varying, dense cost function.

(d) n_x varying, diagonal cost function.

Figure 9.6: Comparison of Hessian condensing and factorization algorithms for MHE: Algorithm 20 + $\mathcal{O}(N^3)$ Cholesky factorization (corresponding to the MPC Algorithm 9 + $\mathcal{O}(N^3)$ Cholesky factorization with cost $\mathcal{O}(N^3)$ and $\mathcal{O}(n_x^2)$) (blue), Algorithm 26 (corresponding to the MPC Algorithm 15 with cost $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^2)$) (red), Algorithm 27 (corresponding to the MPC Algorithm 16 with cost $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^3)$) (green). The time to compute Γ_v^T is included in the total solution time.

9.2.4.2 $\mathcal{O}(N)$ solution

If the $\mathcal{O}(N)$ factorization Algorithm is employed to compute the lower triangular Cholesky factor $\hat{\mathcal{L}}_R$ of the permuted Hessian matrix $\hat{\mathcal{H}}_R$ as in Algorithms 26 or 27, it is possible to exploit the structure still present in $\hat{\mathcal{L}}_R$ to reduce the computational complexity in N of the solution algorithm.

Even more, as shown in the paper [34] for the MPC case, it is not even necessary to explicitly build the lower Cholesky factor $\hat{\mathcal{L}}_R$, reducing the factorization cost by $N^2 n_x n_u^2 + 2N n_x^2 n_u$ flops (and e.g. making Algorithm 27 linear in N). In fact, only the matrices Λ_i, L_i and $\mathcal{L}_0$ are employed in the solution algorithm.

The lower Cholesky factor $\hat{\mathcal{L}}_R$ in (9.37) has the structure

$$\hat{\mathcal{L}}_R = \left[\begin{array}{ccc|c} \Lambda_2 & & & \\ B_1^T L_2^T & \Lambda_1 & & \\ B_0^T A_1^T L_2^T & B_0^T L_1^T & \Lambda_0 & \\ \hline A_0^T A_1^T L_2^T & A_0^T L_1^T & L_0^T & P_0 \end{array}\right] = \hat{\Lambda} + \hat{\Gamma}_u^T \hat{L}^T = \hat{\Lambda} + \hat{B}^T \hat{A}^{-T} \hat{L}^T$$

where

$$\hat{\Lambda} = \begin{bmatrix} \Lambda_2 & & & \\ & \Lambda_1 & & \\ & & \Lambda_0 & \\ & & & \mathcal{L}_0 \end{bmatrix}, \quad \hat{L}^T = \begin{bmatrix} L_2^T & & & \\ & L_1^T & & \\ & & L_0^T & \end{bmatrix},$$

$$\hat{B}^T = \begin{bmatrix} B_2^T & & & \\ & B_1^T & & \\ & & B_0^T & \\ & & & I \end{bmatrix}, \quad \hat{A}^{-T} = \begin{bmatrix} I & & & \\ -A_2^T & I & & \\ & -A_1^T & I & \\ & & -A_0^T & I \end{bmatrix}^{-1}.$$

By defining the vector $\hat{y} = \hat{\mathcal{L}}_R^T \hat{u}$, the forward substitution is in the form

$$\hat{\mathcal{L}}_R \hat{y} = \left(\hat{\Lambda} + \hat{B}^T \hat{A}^{-T} \hat{L}^T\right) \hat{y} = -\hat{\gamma}_r$$

that gives y with the recursion

$$\hat{y} = -\hat{\Lambda}^{-1} \left(\hat{\gamma}_r + \hat{B}^T \hat{A}^{-T} \hat{L}^T \hat{y}\right)$$

that for $N = 3$ looks like

$$\begin{bmatrix} y_3 \\ y_2 \\ y_1 \\ y_0 \end{bmatrix} = \begin{bmatrix} -\Lambda_2^{-1} (\gamma_3) \\ -\Lambda_1^{-1} \left(\gamma_2 + B_1^T L_2^T y_3\right) \\ -\Lambda_0^{-1} \left(\gamma_1 + B_0^T A_1^T L_2^T y_3 + B_0^T L_1^T y_2\right) \\ -\mathcal{L}_0^{-1} \left(\gamma_0 + A_0^T A_1^T L_2^T y_3 + A_0^T L_1^T y_2 + L_0^T y_1\right) \end{bmatrix}.$$

Notice that this is a backward recursion with respect to the indexes of the data matrices, due to the permutation of the condensed Hessian.

The backward substitution is in the form

$$\hat{\mathcal{L}}_R^T \hat{u} = \left(\hat{\Lambda}^T + \hat{L}\hat{A}^{-1}\hat{B} \right) \hat{u} = \hat{y}$$

that gives the recursion

$$\hat{u} = \hat{\Lambda}^{-T} \left(\hat{y} - \hat{L}\hat{A}^{-1}\hat{B}\hat{u} \right)$$

that for $N = 3$ looks like

$$\begin{bmatrix} u_2 \\ u_1 \\ u_0 \\ x_0 \end{bmatrix} = \begin{bmatrix} \Lambda_2^{-T} \left(y_3 - L_2 B_1 u_1 - L_2 A_1 B_0 u_0 - L_2 A_1 A_0 x_0 \right) \\ \Lambda_1^{-T} \left(y_2 - L_1 B_0 u_0 - L_1 A_0 x_0 \right) \\ \Lambda_0^{-T} \left(y_1 - L_0 x_0 \right) \\ \mathcal{L}_0^{-T} \left(y_0 \right) \end{bmatrix}.$$

Notice that this is a forward recursion with respect to the indexes of the data matrices, due to the permutation of the condensed Hessian.

The algorithm is summarized in Algorithm 28. The computational cost of the algorithm is of $4Nn_x^2 + 8Nn_x n_u + 2Nn_u^2$ flops, plus enabling a reduction of the cost of the factorization Algorithms 26 and 27 of $N^2 n_x n_u^2 + 2Nn_x^2 n_u$ flops compared with the $\mathcal{O}(N^2)$ solution algorithm.

9.3 Conclusion

This chapter presented and compared in a systematic way several condensing algorithms, for both the MPC and the MHE problem. Namely, three condensing algorithms, two factorization algorithms, two combined condensing-factorization algorithms and finally two solution algorithms.

Regarding condensing algorithms, in the MPC case, the three condensing algorithms have different asymptotic computational complexities, of $\mathcal{O}(N^3)$ and $\mathcal{O}(n_x^2)$, of $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^2)$ and of $\mathcal{O}(N^2)$ and $\mathcal{O}(n_x^3)$ flops. The former algorithm is the classical condensing algorithm and it is the best option for small values of the ratio N/n_x. The latter algorithm employs a recursion somehow analogue to the Riccati recursion, and it is the best option for large values of the ratio N/n_x. The middle algorithm has the best computational complexity but larger coefficients, and it is therefore always the second best. In the MHE case, all condensing algorithms have a complexity $\mathcal{O}(n_x^3)$, and therefore the latter algorithm is generally the best choice for all problem sizes. Notice that in both

Algorithm 28 Computation of the solution of the condensed system, $\mathcal{O}(N)$ algorithm

Require:
Λ_i, L_i, $\mathcal{L}_0$

$t_N \leftarrow 0$
$y_N \leftarrow -\Lambda_{N-1}^{-1}(\gamma_N)$
for $i \leftarrow N-1, \ldots, 1$ **do**
  $t_i \leftarrow L_i^T y_{i+1} + A_i^T t_{i+1}$
  $y_i \leftarrow -\Lambda_{i-1}^{-1}\left(\gamma_i + B_{i-1}^T t_i\right)$
end for
$t_0 \leftarrow L_0^T y_1 + A_0^T t_1$
$y_0 \leftarrow -\mathcal{L}_0^{-1}(\gamma_0 + t_0)$
$x_0 \leftarrow \mathcal{L}_0^{-T}(y_0)$
$t_0 \leftarrow x_0$
for $i \leftarrow 0, \ldots, N-1$ **do**
  $u_i \leftarrow \Lambda_i^{-T}(y_i - L_i t_i)$
  $t_{i+1} \leftarrow B_i u_i + A_i t_i$
end for

the MPC and the MHE case it is possible to compute the condensed Hessian matrix in time $\mathcal{O}(N^2)$.

Regarding factorization algorithms, in the MPC case, the two factorization algorithms are the classical Cholesky factorization of the condensed Hessian matrix (of size Nn_u) with computational cost $\mathcal{O}((Nn_u)^3)$ (and therefore cubic in N and constant in n_x) and a structure-exploiting factorization with computational cost of $\mathcal{O}(N(n_x^2 n_u + n_x n_u^2 + n_u^3))$ (and therefore linear in N and quadratic in n_x). In the MHE case, the condensed Hessian matrix has size $n_x + Nn_u$, and therefore the classical Cholesky factorization has a computational cost of $\mathcal{O}((n_x + Nn_u)^3)$, that is always larger than the computational cost of the structure-exploiting factorization (equal to $\mathcal{O}(n_x^3 + N(n_x^2 n_u + n_x n_u^2 + n_u^3))$ flops in the MHE case).

The structure-exploiting factorization needs to be combined with either the middle or the latter condensing algorithms. The resulting algorithm has approximately the same computational complexity as the condensing algorithm alone, thanks to the low computational complexity of the $\mathcal{O}(N)$ structure-exploiting factorization. In particular, in both the MPC and the MHE case it is possible to compute the Cholesky factorization of the condensed Hessian matrix in time $\mathcal{O}(N^2)$.

Regarding the solution algorithms, it is possible to solve the KKT system with

the classical backward-forward substitution employing the Cholesky factorization of the condensed Hessian at a computational cost of $\mathcal{O}(N^2)$ flops, or it is possible to employ a structure-exploiting solution at a computational cost of $\mathcal{O}(N)$ flops.

Notice that the use of the structure-exploiting solution removes the need to explicitly build the condensed Hessian matrix, reducing the computational cost of the condensing algorithms. Therefore, it is possible to solve the KKT system in time $\mathcal{O}(N)$ by combining the latter condensing algorithm stopped before the explicit Hessian build, the structure-exploiting factorization and the structure-exploiting solution. The resulting algorithm has analogies with the backward Riccati recursion, and they share the same computational complexity.

On the other hand, if the condensed Hessian matrix or its Cholesky factor are explicitly needed, the $\mathcal{O}(N^2)$ cost can not be avoided in general, trivially because the condensed Hessian matrix contains $\mathcal{O}(N^2)$ elements.

Chapter 10

Partial condensing

The paper [18] proposes techniques to control the level of sparsity in MPC problems, trading-off horizon length and input vector size. The introduction of this degree of freedom in the formulation of the MPC problem can be used to find the optimal trade-off, minimizing the flop count for the MPC solver. Namely, the horizon length can be reduced at the expense of a larger input vector size (partial condensing), or conversely the input vector size can be reduced at the expense of a larger horizon length (sequential update). If roughly $n_x > n_u$, it is found that a decrease of the horizon length leads to a reduction in the flop count, while if roughly $n_x < n_u$, an increase of the horizon length leads to a reduction in the flop count.

In this chapter, it is considered only the partial condensing case, i.e. the decrease of the horizon length at the expense of augmenting the input vector size, even if some of the results about the optimal horizon length selection cover the sequential update case as well. Moreover, the influence of linear algebra routines implementation on the performance of partial condensing is investigated. Therefore, partial condensing is considered as a technique to reduce the solution time of unconstrained linear MPC (and MHE) problems, disregarding the effect on inequality constraints.

The idea of partial condensing can be introduced through an example. Let us consider a MPC problem with horizon length $N = 6$, and let us define the

partition of the state and input vectors such that the states and inputs of $N_c = 3$ consecutive stages are grouped together into blocks (where a block is defined as a group of consecutive stages)

$$\bar{x} = \left[\begin{array}{c} x_0 \\ x_1 \\ x_2 \\ \hline x_3 \\ x_4 \\ x_5 \\ \hline x_6 \end{array}\right] = \begin{bmatrix} \bar{x}_0 \\ \bar{x}_3 \\ x_6 \end{bmatrix}, \quad \bar{u} = \left[\begin{array}{c} u_0 \\ u_1 \\ u_2 \\ \hline u_3 \\ u_4 \\ u_5 \end{array}\right] = \begin{bmatrix} \bar{u}_0 \\ \bar{u}_3 \end{bmatrix}.$$

These can be seen as the state and input vectors of a MPC problem with horizon length $N_p = N/N_c = 2$.

The cost function matrices and vectors can be partitioned accordingly, as

$$\bar{Q} = \left[\begin{array}{ccc|c|c} Q_0 & & & & \\ & Q_1 & & & \\ & & Q_2 & & \\ \hline & & & Q_3 & \\ & & & & Q_4 \\ & & & & & Q_5 \\ \hline & & & & & & Q_6 \end{array}\right] = \begin{bmatrix} \bar{Q}_0 & 0 & 0 \\ 0 & \bar{Q}_3 & 0 \\ 0 & 0 & Q_6 \end{bmatrix}, \quad \bar{q} = \left[\begin{array}{c} q_0 \\ q_1 \\ q_2 \\ \hline q_3 \\ q_4 \\ q_5 \\ \hline q_6 \end{array}\right] = \begin{bmatrix} \bar{q}_0 \\ \bar{q}_3 \\ q_6 \end{bmatrix},$$

$$\bar{S} = \left[\begin{array}{ccc|c|c} S_0 & & & & \\ & S_1 & & & \\ & & S_2 & & \\ \hline & & & S_3 & \\ & & & & S_4 \\ & & & & & S_5 \end{array}\right] = \begin{bmatrix} \bar{S}_0 & 0 & 0 \\ 0 & \bar{S}_3 & 0 \end{bmatrix},$$

$$\bar{R} = \left[\begin{array}{ccc|c|c} R_0 & & & & \\ & R_1 & & & \\ & & R_2 & & \\ \hline & & & R_3 & \\ & & & & R_4 \\ & & & & & R_5 \end{array}\right] = \begin{bmatrix} \bar{R}_0 & 0 \\ 0 & \bar{R}_3 \end{bmatrix}, \quad \bar{r} = \left[\begin{array}{c} r_0 \\ r_1 \\ r_2 \\ \hline r_3 \\ r_4 \\ r_5 \end{array}\right] = \begin{bmatrix} \bar{r}_0 \\ \bar{r}_3 \end{bmatrix}.$$

Therefore the cost function expression can be rewritten as

$$\begin{aligned}
\phi &= \left(\sum_{n=0}^{N-1} \phi_n\right) + \phi_N = \\
&= \left(\sum_{n=0}^{N-1} \tfrac{1}{2} u_n^T R_n u_n + u_n^T S_n x_n + \tfrac{1}{2} x_n^T Q_n x_n + r_n^T u_n + q_n^T x_n + \tfrac{1}{2}\rho_n\right) + \phi_N = \\
&= \left(\sum_{i=0}^{N_p-1} \bar{\phi}_{\{k=N_c i\}}\right) + \phi_N = \\
&= \left(\sum_{i=0}^{N_p-1} \tfrac{1}{2} \bar{u}_k^T \bar{R}_k \bar{u}_k + \bar{u}_k^T \bar{S}_k \bar{x}_k + \tfrac{1}{2} \bar{x}_k^T \bar{Q}_k \bar{x}_k + \bar{r}_k^T \bar{u}_k + \bar{q}_k^T \bar{x}_k + \tfrac{1}{2}\bar{\rho}_k\right) + \phi_N
\end{aligned}$$

where $k = N_c i$ and

$$\phi_N = \left(\tfrac{1}{2} x_N^T Q_N x_N + q_N^T x_N + \tfrac{1}{2}\rho_N\right).$$

The idea of partial condensing is to remove all but the first state component in each block. This can be done by means of condensing. More precisely, since the first state component is retained as an optimization variable, the condensing within each block is analogue to the condensing for the MHE problem.

Namely, given an horizon within each block (or block size) of $N_c = 3$, the state vector of each block $\bar{x}_k$ can be computed as function of the initial state x_k and the inputs $\bar{u}_k$ of the block, as

$$\bar{x}_k = \Gamma_{x,k} x_k + \Gamma_{u,k} \bar{u}_k + \Gamma_{b,k} \tag{10.1}$$

where

$$\Gamma_{x,k} = \begin{bmatrix} I \\ A_k \\ A_{k+1}A_k \end{bmatrix}, \quad \Gamma_{u,k} = \begin{bmatrix} 0 & 0 & 0 \\ B_k & 0 & 0 \\ A_{k+1}B_k & B_{k+1} & 0 \end{bmatrix}, \quad \Gamma_{b,k} = \begin{bmatrix} 0 \\ b_k \\ A_{k+1}b_k + b_{k+1} \end{bmatrix}.$$

The initial state of the following block $x_{k+N_c} = x_{k+3}$ can be computed as a function of the initial state x_k and the inputs $\bar{u}_k$ of the current block as

$$x_{k+3} = \bar{A}_k x_k + \bar{B}_k \bar{u}_k + \bar{b}_k. \tag{10.2}$$

where

$$\begin{aligned}
\bar{A}_k &= A_{k+2}A_{k+1}A_k \\
\bar{B}_k &= \begin{bmatrix} A_{k+2}A_{k+1}B_k & A_{k+2}B_{k+1} & B_{k+2} \end{bmatrix} \\
\bar{b}_k &= A_{k+2}(A_{k+1}b_k + b_{k+1}) + b_{k+2}.
\end{aligned}$$

This equation is the state-space equation of a MPC problem with horizon $N_p = 2$, that has the same state vector size n_x and larger input vector size $N_c n_u$.

Combining the expressions (10.1) and (10.2) gives

$$\left[\begin{array}{c} \bar{x}_k \\ \hline x_{k+3} \end{array}\right] = \Gamma_x x_k + \Gamma_u \bar{u}_k + \Gamma_b$$

where

$$\Gamma_x = \left[\begin{array}{c} I \\ A_k \\ A_{k+1}A_k \\ \hline A_{k+2}A_{k+1}A_k \end{array}\right], \quad \Gamma_u = \left[\begin{array}{ccc} 0 & 0 & 0 \\ B_k & 0 & 0 \\ A_{k+1}B_k & B_{k+1} & 0 \\ \hline A_{k+2}A_{k+1}B_k & A_{k+2}B_{k+1} & B_{k+2} \end{array}\right],$$

$$\Gamma_b = \left[\begin{array}{c} 0 \\ b_k \\ A_{k+1}b_k + b_{k+1} \\ \hline A_{k+2}(A_{k+1}b_k + b_{k+1}) + b_{k+2} \end{array}\right].$$

These expressions clearly resemble the expressions for $\Gamma_v = \left[\Gamma_x \mid \Gamma_u\right]$ and Γ_b in (9.25).

By inserting (10.1) in the cost function, it is possible to rewrite it as a function of the initial state x_k and of the inputs $\bar{u}_k$ of each block. Namely, the cost function at each block becomes

$$\begin{aligned}
&\bar{\phi}_k = \\
&= \frac{1}{2}\begin{bmatrix} x_k \\ \bar{u}_k \end{bmatrix}^T \begin{bmatrix} \Gamma_{x,k}^T \bar{Q}_k \Gamma_{x,k} & \Gamma_{x,k}^T \bar{Q}_k \Gamma_{u,k} + \Gamma_{x,k}^T \bar{S}_k^T \\ \Gamma_{u,k}^T \bar{Q}_k \Gamma_{x,k} + \bar{S}_k \Gamma_{x,k} & \Gamma_{u,k}^T \bar{Q}_k \Gamma_{u,k} + \Gamma_{u,k}^T \bar{S}_k^T + \bar{S}_k \Gamma_{u,k} + \bar{R}_k \end{bmatrix} \begin{bmatrix} x_k \\ \bar{u}_k \end{bmatrix} + \\
&+ \begin{bmatrix} \Gamma_{x,k}^T \bar{Q}_k \Gamma_{b,k} + \Gamma_{x,k}^T \bar{q}_k \\ \Gamma_{u,k}^T \bar{Q}_k \Gamma_{b,k} + \bar{S}_k \Gamma_{b,k} + \Gamma_{u,k}^T \bar{q}_k + \bar{r}_k \end{bmatrix}^T \begin{bmatrix} x_k \\ \bar{u}_k \end{bmatrix} + \frac{1}{2}\left(\Gamma_{b,k}^T \bar{Q}_k \Gamma_{b,k} + 2\bar{q}_k^T \Gamma_{b,k} + \bar{\rho}_k\right) = \\
&= \frac{1}{2}\begin{bmatrix} x_k \\ \bar{u}_k \end{bmatrix}^T \begin{bmatrix} H_{Q,k} & H_{S,k}^T \\ H_{S,k} & H_{R,k} \end{bmatrix} \begin{bmatrix} x_k \\ \bar{u}_k \end{bmatrix} + \begin{bmatrix} g_{q,k} \\ g_{r,k} \end{bmatrix}^T \begin{bmatrix} x_k \\ \bar{u}_k \end{bmatrix} + \rho_{\rho,k}
\end{aligned}$$

The expressions for $H_{Q,k}$, $H_{S,k}$, $H_{R,k}$, $g_{q,k}$, $g_{r,k}$ are formally identical to the expressions in (9.28), provided that $\Gamma_{x,k}$, $\Gamma_{u,k}$, $\Gamma_{b,k}$ are employed in place of Γ_x, Γ_u, Γ_b. In practice, the difference is that Q_{k+N_c} and q_{k+N_c} are considered in the computation of the cost function at the following block, and they should not be considered twice. Therefore, the expressions for $H_{Q,k}$, $H_{S,k}$, $H_{R,k}$, $g_{q,k}$, $g_{r,k}$ can be obtained by employing the condensing algorithms for MHE presented in Section 9.2.1 with $Q_N = 0$ and $q_N = 0$, and possibly by exploiting this to reduce the computational cost.

10.1 Partial condensing algorithms

The condensing algorithms for MHE presented in Section 9.2.1 can be employed for condensing within each block, provided that the terminal cost is set to zero, i.e. $Q_{k+N_c} = 0$ and $q_{k+N_c} = 0$. This can be exploited to reduce the computation cost of the condensing algorithms.

About the choice between the three Hessian condensing algorithms presented in Section 9.2.1, the safest choice is Algorithm 22, that is found to give the best performance in case of free initial state. Furthermore, this condensing algorithm allows to merge the computation of the condensed Hessian matrix and condensed gradient vector in a single routine. A version of this algorithm tailored to the case of terminal cost equal to zero is presented in Algorithm 29.

Algorithm 29 Computation of H_Q, H_S, H_R, g_q, g_r, partial condensing case

Require:
Γ_x, Γ_u, Γ_b

$\begin{bmatrix} D_{N-1} & & \\ M_{N-1}^T & P_{N-1} & \\ m_{N-1}^T & p_{N-1}^T & * \end{bmatrix} \leftarrow \begin{bmatrix} R_{N-1} & & \\ S_{N-1}^T & Q_{N-1} & \\ r_{N-1}^T & q_{N-1}^T & * \end{bmatrix}$
$H_R[N-1, N-1] \leftarrow D_{N-1}$
$H_R[N-1, 0:N-2] \leftarrow M_{N-1} \cdot \Gamma_u[N-1, 0:N-2]$
$H_S[N-1] \leftarrow M_{N-1} \cdot \Gamma_x[N-1]$
$g_r[N-1] \leftarrow m_{N-1} + M_{N-1} \cdot \Gamma_b[N-1]$
for $i \leftarrow N-2, \ldots, 0$ **do**
  $\begin{bmatrix} \mathcal{L}_{00} & \\ \mathcal{L}_{10} & * \end{bmatrix} \leftarrow \begin{bmatrix} P_{i+1} & \\ p_{i+1}^T & * \end{bmatrix}^{1/2}$
  $\mathcal{A}^T\mathcal{L} \leftarrow \begin{bmatrix} B_i^T \\ A_i^T \\ b_i^T \end{bmatrix} \cdot \mathcal{L}_{00} + \begin{bmatrix} \\ \\ \mathcal{L}_{10} \end{bmatrix}$
  $\begin{bmatrix} D_i & & \\ M_i^T & P_i & \\ m_i^T & p_i^T & * \end{bmatrix} \leftarrow \begin{bmatrix} R_i & & \\ S_i^T & Q_i & \\ r_i^T & q_i^T & * \end{bmatrix} + \left(\mathcal{A}^T\mathcal{L}\right) \cdot \left(\mathcal{A}^T\mathcal{L}\right)^T$
  $H_R[i, i] \leftarrow D_i$
  $H_R[i, 0:i-1] \leftarrow M_i \cdot \Gamma_u[i, 0:i-1]$
  $H_S[i] \leftarrow M_i \cdot \Gamma_x[i]$
  $g_r[i] \leftarrow m_i + M_i \cdot \Gamma_b[i]$
end for
$H_Q \leftarrow P_0$
$g_q \leftarrow p_0$

If partial condensing is employed to condense N_c stages, the computational cost of the algorithm is of about $N_c^2 n_x n_u^2 + \frac{7}{3} N_c n_x^3$ flops, plus $N_c^2 n_x^2 n_u + 2 N_c n_x^3$ flops to compute Γ_x, Γ_u and Γ_b.

Let N_p denote the horizon length of the partially condensed MPC (or MHE) problem. If N_p is an exact divisor of N, then each stage of the new MPC problem (in the following, partially condensed MPC problem) is obtained by condensing $N_c = N/N_p$ stages of the original MPC problem. In this case, all stages of the partially condensed MPC problem have an input vector with equal size $N_c n_u$, where n_u is the input vector size of the original MPC problem. On the contrary, if N_p is not an exact divisor of N, then different stages of the partially condensed MPC problem can have different input vector size. Therefore MPC solvers supporting stage-variant n_u and n_x are needed to solve the partially condensed MPC problem.

Assuming that N_c is an integer divisor of N and that $N = N_c \cdot N_p$, the partially condensed MPC (or MHE) problem has N_p stages. If the MPC (or MHE) problem is time-invariant, then the cost function matrices of the partially condensed problem are time-invariant as well, and therefore the partial condensing algorithm needs to be executed only once. If the MPC (or MHE) problem is time-variant, then the partial condensing algorithm needs to be executed N_p times, once per block.

10.2 Choice of N_p

This section presents a method to find the theoretically best choice for the trade-off between horizon length and input vector size. The analysis assumes that the partially condensed MPC problem is solved by means of the backward Riccati recursion presented in Section 8.1. Only the MPC case is considered, the MHE case being analogue.

The computational cost of the backward Riccati recursion is of

$$\left(\tfrac{1}{3} n_x^3\right) + (N-1)\left(\tfrac{7}{3} n_x^3 + 4 n_x^2 n_u + 2 n_x n_u^2 + \tfrac{1}{3} n_u^3\right) + \left(n_x^2 n_u + n_x n_u^2 + \tfrac{1}{3} n_u^3\right) \quad (10.3)$$

flops, where terms due to the last, middle and first stages are grouped together.

The following analysis requires some assumption.

Assumption 1 N is large, such that the computational cost can be approximated as

$$N\left(\tfrac{7}{3}n_x^3 + 4n_x^2 n_u + 2n_x n_u^2 + \tfrac{1}{3}n_u^3\right) = n_x^3 N\left(\tfrac{1}{3m^3} + \tfrac{2}{m^2} + \tfrac{4}{m} + \tfrac{7}{3}\right)$$

where

$$m = \frac{n_x}{n_u}$$

is the ratio between the number of states and the number of inputs.

Assumption 2 the horizon length and the input vector size can be traded off continuously. This means that the horizon length in the partially condensed problem can be chosen as $N_p = \frac{N}{\alpha}$ for some $\alpha \in \mathbb{R}$, $\alpha > 0$, and consequently the input vector size is αn_u. Therefore, the computational cost to solve the partially condensed problem using the backward Riccati recursion is of

$$n_x^3 \frac{N}{\alpha}\left(\tfrac{1}{3m^3}\alpha^3 + \tfrac{2}{m^2}\alpha^2 + \tfrac{4}{m}\alpha + \tfrac{7}{3}\right)$$

flops, and it is a function of α. Notice that $\alpha > 1$ corresponds to a reduction of the horizon length (the case covered here), while $\alpha < 1$ corresponds to an increase of the horizon length (the case not covered here). The minimizer of the function can be found by setting its derivative to zero. For $\alpha > 0$, the minimum of this function is attained for

$$\alpha = 0.94224m \quad \Rightarrow \quad N_p \approx \frac{N}{0.94224m} \approx \frac{N}{m}$$

and it is a function of the ratio m only. In particular, for $n_x \approx n_u$, there is no advantage in reducing or augmenting the horizon length.

The approximate maximum flops reduction factor in solving the MPC problem is

$$f(m) = \frac{n_x^3 \frac{N}{\alpha}\left(\frac{1}{3m^3}\alpha^3 + \frac{2}{m^2}\alpha^2 + \frac{4}{m}\alpha + \frac{7}{3}\right)}{n_x^3 N\left(\frac{1}{3m^3} + \frac{2}{m^2} + \frac{4}{m} + \frac{7}{3}\right)} = \frac{8.6568m^2}{\left(\frac{7}{3}m^3 + 4m^2 + 2m + \frac{1}{3}\right)}$$

and it is a function of m only.

The function $f(m)$ is plotted for $0.01 < m < 100$ in Figure 10.1. Since $\alpha \approx m$, the maximum flop reduction is obtained by reducing the horizon length for $m < 1$ and by increasing it for $m > 1$. Furthermore, the maximum flop reduction is rather small for $n_u \approx n_x$, and it gets significant only if n_x and n_u are of rather different size.

In practice, the best value of N_p is affected by the value of N, since a small value of N limits the number of possible choices for N_p. Furthermore, the different

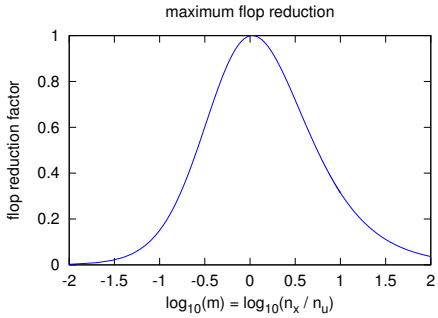


Figure 10.1: Maximum flop reduction factor in choosing the optimal N vs n_u trade-off.

performance of the linear algebra routines for different matrix sizes affects the practical reduction in computational time, as shown in the next section.

10.3 Influence of linear algebra routines performance

This section investigates the influence of linear algebra routines performance on the optimal choice for N_p.

The performance of linear algebra routines generally depends on implementation choices and matrix sizes, as widely shown in Part I of this thesis. In particular, for linear algebra routines in HPMPC, the performance increases quickly for matrix size up to about 20-50, and then it stabilizes close to the peak performance for matrices of size up to about 300-400, before slightly decreasing due to lack of blocking for cache. The exact intervals depend on the ISA and on the specific linear algebra routine. Routines generally can attain only a small fraction of the peak throughput for very small matrices, limiting the solver performance in case of small-scale MPC problems. Partial condensing is therefore expected to further improve performance beyond the flop reduction, due to the use of larger matrices.

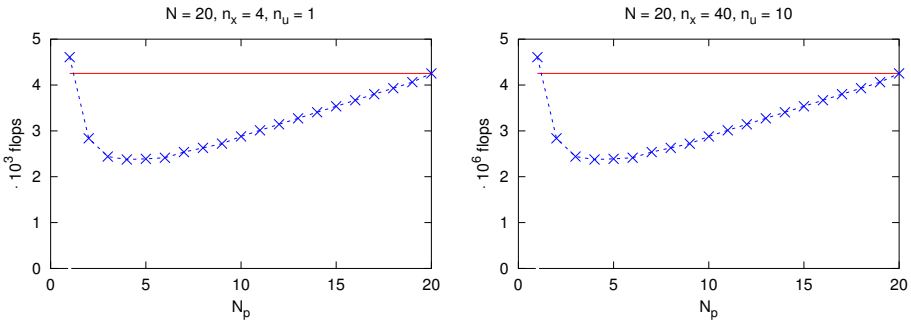
Figure 10.2 investigates this through an example. The test processor is the Intel Core i7-4800MQ often considered in this thesis, with compiler `gcc`. Two MPC problems are considered, one small with $n_x = 4$ and $n_u = 1$, and one large with $n_x = 40$ and $n_u = 10$. Both problems have the same horizon length $N = 20$,

and the same ratio $m = {}^{n_x}/_{n_u} = 0.25$. For these values, the analysis in Section 10.2 gives an optimal (real) value of N_p equal to 5.3, and a flop reduction factor of 0.625. These values are valid for both the small-scale and the large-scale problems.

These findings are firstly compared with the reduction in the flop count for the MPC problems for different values of N_p when Assumptions 1 and 2 are dropped. In case N_p is not an integer divisor of N, the input size can be different at different stages of the partially condensed MPC problem: the backward Riccati recursion implementation is designed to handle values of n_x and n_u varying stage-wise. It is clear that figures 10.2a and 10.2b are identical, beside the fact that the y axes is scaled exactly by $1000 = 10^3$ (the computational cost in (10.3) is cubic in n_x and n_u, and the values of the two problems are scaled by a factor 10). The optimal value of N_p is found to be 4, for a reduction factor in the flop count of 0.559. Starting from an N_p value equal $N = 20$ on the right on the plots, the flop count decreases steadily as N_p decreases until it reaches a minimum. For smaller values of N_p, the flop count quickly increases, and for $N_p = 1$ it is larger than the one of the original MPC problem with $N = 20$.

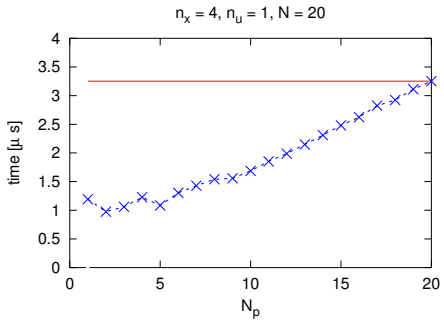
The second row of figures represents the running times, when the linear algebra routines are provided by the C99 target ISA in HPMPC. The size of the `gemm` kernel is 4×4. For the large scale problem, the plot in Figure 10.2d closely resembles the figures from the flop count. However, for the small scale problem, the plot in Figure 10.2c is rather different: the best value for N_p is 2, and for $N_p = 1$ the solution time is much lower than the original MPC problem. This is due to the fact that for such a small problem size, there is a big advantage in replacing many operations on very small matrices (smaller than the `dgemm` kernel size) with fewer operations with larger matrices, where linear algebra routines can attain a better performance.

This trend gets even more clear in the last row of figures, where the linear algebra routines are provided by the AVX2 target ISA in HPMPC. The optimal kernel size is 12×4, and therefore larger matrices are needed to obtain high performance. For the large scale problem, the plot in Figure 10.2f somehow resembles the figures from the flop count, but the optimal N_p value is 3, and for $N_p = 1$ the solution time is still lower than the original MPC problem (despite the larger number of flops). For the small scale problem, the plot in Figure 10.2e shows a rather steady decrease in the solution time as N_p decreases, with the optimal value attained for $N_p = 1$ (and equal to a reduction in solution time of 0.234, despite an increase in the flop count with respect to the original problem).

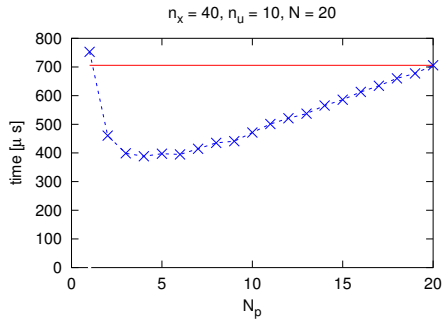


(a) Small-scale, flops.

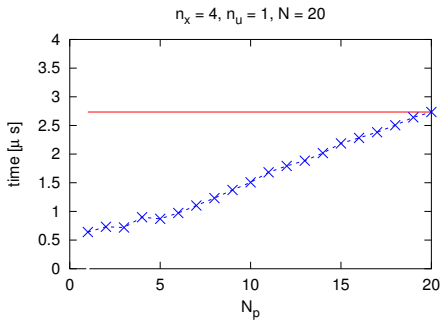
(b) Large-scale, flops.



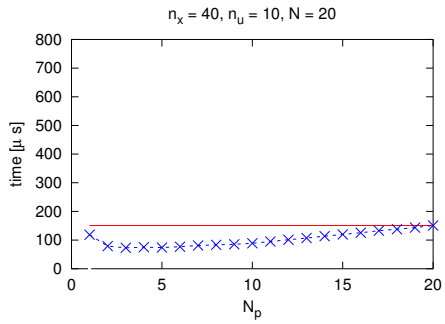
(c) Small-scale, time, C99 ISA.



(d) Large-scale, time, C99 ISA.



(e) Small-scale, time, AVX2 ISA.



(f) Large-scale, time, AVX2 ISA.

Figure 10.2: Flop count / solution time for the reformulations of two MPC problems (small scale one on the left, large scale one on the right), obtained using partial condensing for different values of N_p ($N_p = 20$ is fully sparse, $N_p = 1$ is fully condensed). The flop/time value of the original MPC problem is represented using the red line.

10.4 Conclusion

Partial condensing is a technique that gives a reformulation of the MPC problem, decreasing the horizon length at the expense of an increase in the input vector size. It has been proposed in [18], together with sequential update, consisting in decreasing the input vector size at the expense of increasing the horizon length.

In this chapter, only partial condensing has been considered. By choosing as a solver for the unconstrained MPC problem the backward Riccati recursion in Section 8.1, a simplified analysis shows that the reduction in the flop count is a function only of the ratio $m = {}^{n_x}/_{n_u}$, and gives a simple formula to compute the theoretical optimal horizon length N_p and the corresponding flop reduction factor. Numerical tests confirm the validity of the analysis, in the case of linear algebra routines giving steady performance for all matrix sizes of interest.

In practice, in case of small matrices the performance of linear algebra routines increases with the matrix size, and therefore for small-scale MPC problems the optimal value of N_p is smaller than the one minimizing the flop count, since the use of larger matrices boosts performance. In this sense, partial condensing is an extremely interesting technique, allowing to better achieve high FP throughput and obtain high performance also in case of small-scale MPC problems.

Chapter 11

Unconstrained MPC problems with time-invariant matrices

This chapter presents tailored algorithms for the solution of unconstrained MPC problems where all matrices of cost function and state-space system are time-invariant, while all vectors are time variant, in the special case where the arrival cost is given by the solution of the Discrete Algebraic Riccati Equation (DARE).

This problem formulation is a special instance of (7.1), and therefore it can be solved using e.g. the backward Riccati recursion presented in Section 8.1 in time $\mathcal{O}(N(n_x + n_u)^3)$, or one of the condensing methods presented in Section 9.1. Generally speaking, since all matrices are time-invariant, the KKT matrix can be factorized off-line. However, the fact that the Hessian of the arrival cost is the solution of the DARE can be exploited to factorize the KKT matrix in time constant in N. This reduces the solution time to $\mathcal{O}(N(n_x+n_u)^2)$. Furthermore, the memory usage of the solution algorithms can be greatly reduced, further improving performance of the level 2 BLAS linear algebra routines employed in the solution algorithm.

Formally, the results of this chapter could be extended to the MHE problem. However, it does not make sense to consider a MHE problem with $Q_0 = Q$

and $Q_N = P_\infty$. Therefore, the MHE problem is not further mentioned in this chapter.

11.1 Problem formulation

The unconstrained MPC problem with Time-Invariant Matrices (MPC-TIM) is defined as

$$\min_{u,x} \quad \sum_{n=0}^{N-1} \frac{1}{2} \begin{bmatrix} u_n \\ x_n \end{bmatrix}^T \begin{bmatrix} R & S \\ S^T & Q \end{bmatrix} \begin{bmatrix} u_n \\ x_n \end{bmatrix} + \begin{bmatrix} r_n \\ q_n \end{bmatrix}^T \begin{bmatrix} u_n \\ x_n \end{bmatrix} + \frac{1}{2} x_N^T P_\infty x_N + q_N^T x_N \tag{11.1a}$$

$$s.t. \quad x_{n+1} = Ax_n + Bu_n + b_n \tag{11.1b}$$

$$x_0 = \hat{x}_0 \tag{11.1c}$$

where all matrices in the dynamical system and the cost function are time-invariant, while the vectors can be time-variant. The optimization is performed over the finite horizon N, and $\begin{bmatrix} R & S \\ S^T & Q \end{bmatrix} \geq 0$, $R > 0$ and $P \geq 0$. The Hessian of the arrival cost is initialized to the solution P_∞ of the DARE,

$$P_\infty = Q + A^T P_\infty A - (S^T + A^T P_\infty B)(R + B^T P_\infty B)^{-1}(S + B^T P_\infty A). \tag{11.2}$$

11.2 Motivation

Problems in the form of the MPC-TIM (11.1) are encountered in a number of situations. The choice $P = P_\infty$ is either required from or compatible with all these cases, and it is the key to improve the performance of the solution methods.

11.2.1 Linear time-invariant control problems

Problems in the form (11.1) arise in the (unconstrained) predictive control of linear systems [52]. In this class of problems, the state-space and the cost function matrices are time-invariant. However, the vectors can be time-variant. The terms q_n and r_n can e.g. be employed to predictively handle changes in the set point. The term b_n can be employed to predictively handle changes in known disturbances, as $b_n = Ed_n$. Alternatively, the term b_n arises in the control of state space systems in innovation form [53], as $b_0 = Ke_n$.

11.2.2 Sub-problem in splitting methods for linear MPC

The linear time-invariant (finite horizon) MPC problem

$$\begin{aligned}
\min_{u,x} \quad & \sum_{n=0}^{N-1} \tfrac{1}{2} x_n^T Q x_n + \tfrac{1}{2} u_n^T R u_n + \tfrac{1}{2} x_N^T P_\infty x_N && \text{(11.3a)}\\
s.t. \quad & x_{n+1} = A x_n + B u_n && \text{(11.3b)}\\
& x_0 = \hat{x}_0 && \text{(11.3c)}\\
& C_x x_n \leq c_x && \text{(11.3d)}\\
& C_u x_n \leq c_u && \text{(11.3e)}
\end{aligned}$$

can be solved by applying slitting methods such as the Alternating Minimization Algorithm (AMA) to the dual of the problem [82]. At each iteration of the splitting method, the input u_n and state x_n sequences are obtained by solving a problem in the form

$$\begin{aligned}
\min_{u,x} \quad & \sum_{n=0}^{N-1} \tfrac{1}{2} u_n^T R u_n + \tfrac{1}{2} x_n^T Q x_n - \begin{bmatrix} C_x x_n - c_x \\ C_u u_n - c_u \end{bmatrix}^T \lambda_n + \frac{1}{2} x_N^T P_\infty x_N && \text{(11.4a)}\\
s.t. \quad & x_{n+1} = A x_n + B u_n && \text{(11.4b)}\\
& x_0 = \hat{x}_0 && \text{(11.4c)}
\end{aligned}$$

(i.e. in the form of the MPC-TIM (11.1)) where λ_n are not optimization variables in (11.4). The advantage of the use of AMA over other splitting methods such as the Alternating Direction Method of Multipliers (ADMM) is that the Hessian of the cost function in (11.4) is the same as the original MPC problem, and therefore P is still initialized to the solution of the DARE.

11.2.3 Sub-problem in splitting methods for constrained LQR

In the papers [80, 81] authors propose to solve the (infinite horizon) Constrained Linear-Quadratic Regulator (CLQR) problem

$$\min_{u,x} \quad \sum_{n=0}^{\infty} \tfrac{1}{2} x_n^T Q x_n + \tfrac{1}{2} u_n^T R u_n \tag{11.5a}$$

$$s.t. \quad x_{n+1} = A x_n + B u_n \tag{11.5b}$$

$$x_0 = \hat{x}_0 \tag{11.5c}$$

$$C_x x_n \leq c_x \tag{11.5d}$$

$$C_u x_n \leq c_u \tag{11.5e}$$

by applying splitting methods such as Alternating Minimization Algorithm (AMA) and Forward-Backward Splitting (FBS) to the dual of the problem. Under assumptions clearly stated on the papers, the proposed algorithmic scheme requires a finite amount of computations: if the u_n and x_n sequences last hit a constraints at stage T^k, the sequence of the Lagrangian multipliers associated with the inequality constraints gets zero for all stages $n > T^k$.

For $n > T^k$ the problem reduces to the unconstrained LQR [56], and the optimal input sequence can be computed as the time-invariant state feedback

$$u_n = K_\infty x_n$$

where in this case the LQ-gain is

$$K_\infty = -(R + B^T P_\infty B)^{-1}(B^T P_\infty A)$$

and P_∞ is the solution of the Discrete Algebraic Riccati Equation (DARE) associated with the matrices A, B, Q, R:

$$P_\infty = Q + A^T P_\infty A - (B^T P_\infty A)^T (R + B^T P_\infty B)^{-1}(B^T P_\infty A).$$

For $n \leq T^k$, the input u_n and state x_n sequences are obtained by solving a problem in the form

$$\min_{u,x} \quad \sum_{n=0}^{N-1} \tfrac{1}{2} u_n^T R u_n + \tfrac{1}{2} x_n^T Q x_n - \begin{bmatrix} C_x x_n - c_x \\ C_u u_n - c_u \end{bmatrix}^T \lambda_n + \frac{1}{2} x_N^T P_\infty x_N \tag{11.6a}$$

$$s.t. \quad x_{n+1} = A x_n + B u_n \tag{11.6b}$$

$$x_0 = \hat{x}_0 \tag{11.6c}$$

(that is formally identical to (11.4)) at each iteration of the splitting method, where λ_n are not optimization variables in (11.6). The key difference with respect to the MPC case (i.e. the case with finite horizon) is that the horizon length $N = T^k$ can increase with the iteration count k of the splitting method. If the assumption $P = P_\infty$ is dropped, in general problems in the form (11.1) if the horizon length N increases the factorization of the sparse KKT system or of the condensed Hessian needs to be updated, or computed off-line for a sufficiently long horizon N. However, the fact that $P = P_\infty$ in (11.6) can be exploited to completely avoid the factorization of the sparse KKT matrix or of the condensed Hessian.

11.3 Sparse formulation

If problem (11.1) is considered in the sparse formulation where both states x and inputs u are retained as optimization variables, then the natural choice is its solution using the backward Riccati recursions. In fact, the backward Riccati recursion can be seen as a stage-wise factorization of the KKT matrix [70].

The key idea is that the backward Riccati recursion

$$P_n = Q + A^T P_{n+1} A - (S + B^T P_{n+1} A)^T (R + B^T P_{n+1} B)^{-1} (S + B^T P_{n+1} A)$$

reduces to the DARE (11.2) if $P = P_\infty$. This means that all blocks of the stage-wise factorization of the KKT matrix are constant over the stages, and therefore can be computed off-line. As a further advantage, only the matrices of one stage need to be stored, making the memory requirement for the factorized KKT matrix constant with respect to the horizon length N. This fact can be exploited to design algorithms to solve the problem (11.1) without the need to explicitly build or factorize the sparse KKT matrix (or condensed Hessian, in the next section).

Since the linear terms in the dynamic equations and cost function are time-variant, a backward vector recursion has to be performed on-line. The optimal value of u_n is therefore computed as the affine state feedback

$$u_n = K_\infty x_n + k_n$$

where K_∞ is the LQ-gain, computed off-line using the P_∞ matrix as

$$K_\infty = -(R + B^T P_\infty B)^{-1} (S + B^T P_\infty A), \tag{11.7}$$

while k_n is computed on-line as

$$k_n = -(R + B^T P_\infty B)^{-1} (r_n + B^T (P_\infty b_n + p_{n+1})),$$

and p_n is computed by means of the backward recursion

$$p_n = q_n + A^T p_{n+1} + K_\infty^T (r_n + B^T (P_\infty b_n + p_{n+1})).$$

The algorithm is presented in Algorithm 30. The computational cost of the algorithm is of

$$N(6n_x^2 + 8n_x n_u + 2n_u^2)$$

flops, that can be reduced by $2n_x^2$ flops if b_n is time invariant and the product $P_\infty \cdot b$ is computed off-line. Beside the storing of the u_n and x_n sequences, the working memory requirements for the algorithm are only the Nn_u elements of the k_n sequence.

Algorithm 30 Riccati algorithm for the MPC-TIM problem (11.1)

Require:
P_∞
$D_\infty^{-1} = (R + B^T \cdot P_\infty \cdot B)^{-1}$
$K_\infty = -D_\infty^{-1} \cdot (S + B^T \cdot P_\infty \cdot A)$

1: $p \leftarrow p_N$
2: **for** $n \leftarrow N-1, \ldots, 0$ **do**
3: $\quad p \leftarrow P_\infty \cdot b_n + p$
4: $\quad t \leftarrow r_n + B^T \cdot p$
5: $\quad p \leftarrow q_n + A^T \cdot p + K_\infty^T \cdot t$
6: $\quad k_n \leftarrow -D_\infty^{-1} \cdot t$
7: **end for**
8: **for** $n \leftarrow 0, \ldots, N-1$ **do**
9: $\quad u_n \leftarrow K_\infty \cdot x_n + k_n$
10: $\quad x_{n+1} \leftarrow A \cdot x_n + B \cdot u_n + b_n$
11: **end for**

11.4 Condensed formulation

Condensing techniques have been widely used to speed-up the solution of MPC problem, by reducing the size of the corresponding optimization problem. Namely they exploit the system dynamic equation (11.1b) to rewrite future stats x_n as function of the initial state $\hat{x}_0$ (datum) and of the future inputs u_n (retained as optimization variables). The output of the condensing process is a dense positive-definite system of equations, that has traditionally be considered unstructured and solved by means of Cholesky factorization and forward-backward substitution, at a cost $\mathcal{O}(N^3 n_u^3)$.

The condensing methods presented in Section 9.1 could be employed to solve the MPC-TIM problem (11.1). However, the special structure of the problem can be exploited to reduce the computational cost, similarly to the sparse formulation case.

Namely, the relation between the backward Riccati recursion and the factorization of the condensed Hessian matrix in Algorithm 16 suggests that also in the condensed case the factorization of the entire Hessian matrix can be reduced to the factorization of a single stage.

When the condensed Hessian factorization algorithm 16 is combined with the solution algorithm 17, it is possible to avoid altogether the build of the condensed Hessian matrix or its Cholesky factor.

Therefore, Algorithm 17 can be tailored to the solution of systems of linear equations in the form

$$H\bar{u} = \bar{g} \tag{11.8}$$

where H is the condensed Hessian matrix associated with problem (11.1)

$$\bar{u} = \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-1} \end{bmatrix}, \quad \bar{g} = \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_{N-1} \end{bmatrix}.$$

The fact that in this special case the backward Riccati recursion reduces to the DARE (11.2) implies that there is no need to explicitly build or factorize the Hessian matrix H. In fact, all quantities needed in the backward-forward recursions in Algorithm 17 can be computed off-line using the P_∞ matrix.

The resulting algorithm is presented in Algorithm 31. Line 1 means that the computations can be carried-on in place, overwriting the $\bar{g}$ vector with the solution vector $\bar{u}$. The computational cost of the algorithm is of

$$N(4n_x^2 + 8n_x n_u + 2n_u^2)$$

flops. Beside the storing of the u_n and x_n sequences and the problem data, the algorithm requires a constant amount of working memory (with respect to the horizon length N), equal to the n_x elements of the working vector t.

Algorithm 31 Condensing algorithm for the MPC-TIM (11.1)

Require:

$M_\infty = S + B^T \cdot P_\infty \cdot A$
$D_\infty^{-1} = (R + B^T \cdot P_\infty \cdot B)^{-1}$
$K_\infty = -D_\infty^{-1} \cdot M_\infty$

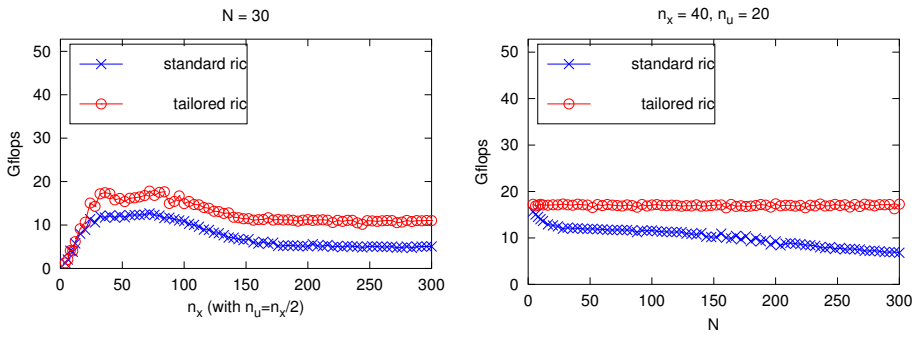
1: $\bar{u} \leftarrow \bar{g}$
2: $u_{N-1} \leftarrow -D_\infty^{-1} \cdot u_{N-1}$
3: $t \leftarrow M_\infty^T \cdot u_{N-1}$
4: $u_{N-2} \leftarrow -D_\infty^{-1} \cdot (u_{N-2} + B^T \cdot t)$
5: **for** $n \leftarrow N-3, \ldots, 0$ **do**
6: $\quad t \leftarrow A^T \cdot t + M_\infty \cdot u_{n+1}$
7: $\quad u_n \leftarrow -D_\infty^{-1} \cdot (u_n + B^T \cdot t)$
8: **end for**
9: $t \leftarrow B \cdot u_0$
10: $u_1 \leftarrow u_1 + K_\infty \cdot t$
11: **for** $n \leftarrow 2, \ldots, N-1$ **do**
12: $\quad t \leftarrow A \cdot t + B \cdot u_{n-1} + b_n$
13: $\quad u_n \leftarrow u_n + K_\infty \cdot t$
14: **end for**

11.5 Implementation aspects

From an implementation point of view, both Algorithms 30 and 31 are very efficient. Both algorithms are implemented using the `gemv` (general matrix-vector multiplication) and `dsymv` (symmetric matrix-vector multiplication) routines (that are part of level 2 BLAS).

Generally speaking, as shown in the first part of this thesis, the performance of level 2 BLAS routines heavily depends on the memory footprint of the data, due to the lack of reuse of matrices elements (or limited reuse, in case of `dsymv`). The performance gets particularly poor if the memory footprint of the control problem data is large enough to exceed cache size, as data needs to be streamed from main memory.

Since in the special case analyzed in the current chapter all matrices in the factorized KKT matrix are time invariant, the amount of memory needed to store them is constant (i.e. does not increase with N), since the same matrices are reused at all stages. This greatly reduces the memory footprint, meaning that the `gemv` routine needed in the implementation can attain its best performance.



(a) n_x. (b) N.

Figure 11.1: Performance plot for Algorithm 17 (blue) and Algorithm 30 (red) in the solution of problem (11.1) once the KKT matrix has been factorized. Test performed on an Intel Core i7 4800MQ processor, `gcc` compiler, AVX2 target ISA in HPMPC.

Figure 11.1 compares the computational performance (in Gflops) of the standard (Algorithm 17) and the tailored (Algorithm 30) Riccati solvers for the solution of the KKT system once it has been factorized. Therefore, this shows the performance advantages given by the reduced memory footprint, beside the reduction in the factorization time. Figure 11.1a shows that the performance of the tailored version is higher for large n_x and n_u, since the matrices of only one stages are needed, thus they can grow larger and still fit in cache. Figure 11.1b shows that the performance of the tailored version does not decrease as N increases, since the same matrices are reused at all stages (making their memory footprint constant with respect to N). This makes this algorithm particularly favorable in case of long horizons.

The results are analogue for the tailored condensing Algorithm 31, and therefore they are not repeated.

11.6 Conclusion

This chapter describes solution methods tailored for the MPC-TIM problem (11.1), that is a special instance of the unconstrained MPC problem (7.1) where all matrices are time-invariant, the arrival cost matrix is initialized to the solution of the DARE, while all vectors can in general be time-variant.

The choice of P implies that the KKT matrix can be factorized in time constant in N, since the the backward Riccati recursion reduces to the DARE, and therefore if the Riccati recursion is used to factorize the KKT matrix all matrices of the factorized KKT matrix are constant stage-wise. The optimal input is computed as an affine state feedback where the time-invariant feedback matrix is the LQ-gain, while the feedback vector is time-varying. Since problems in the form (11.1) arise as sub-problems in splitting methods for the CLQR, this implies that the optimal input sequence of the CLQR can be obtained as an affine state feedback where the feedback matrix is still the LQ-gain.

From an implementation point of view, the fact that the matrices of the factorized KKT matrix are constant stage-wise greatly reduces the required amount of memory, since only the matrices of one stage are needed. As a further advantage, this means that the problem data can fit in cache for much larger values of N (since only vectors require a different instance per each stage), allowing the level 2 BLAS routines to perform well.

As a final note, there is no need to update the KKT matrix factor as the horizon length increases in splitting methods for the CLQR, making the developed algorithms a perfect fit for solving the sub-problems in these methods.

Part III

Algorithms for Constrained and Non-Linear MPC

Chapter 12

Constrained MPC problem formulations

Part III deals with the solution of constrained and non-linear MPC and MHE problems. This last part is much shorter than the first two, and it does not contain novel optimization algorithms for optimal control. The algorithms presented in this part are only meant to be examples of the use of the efficient routines developed in the previous parts of the thesis. The fact that the solvers presented in this section can outperform state-of-the-art solvers for MPC is ultimately due to the superior performance of the routines for linear algebra and unconstrained MPC and MHE problems.

As a difference with respect to part II, in part III only algorithms for MPC are explicitly considered. This choice is justified by the fact that algorithms for MHE are essentially identical, since the specific features of MHE are exploited at the level of the unconstrained sub-problems.

12.1 Linear MPC problem

The linear MPC problem with soft box constraints and hard general constraints is the quadratic program

$$\min_{u,x,s} \quad \sum_{n=0}^{N-1} \frac{1}{2} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix}^T \begin{bmatrix} R_n & S_n & r_n \\ S_n^T & Q_n^T & q_n \\ r_n^T & q_n^T & \rho_n \end{bmatrix} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} x_N \\ 1 \end{bmatrix}^T \begin{bmatrix} Q_N & q_N \\ q_N^T & \rho_N \end{bmatrix} \begin{bmatrix} x_N \\ 1 \end{bmatrix} + \tag{12.1a}$$

$$+ \sum_{n=0}^{N} \frac{1}{2} \begin{bmatrix} s_n \\ 1 \end{bmatrix}^T \begin{bmatrix} Z_n & z_n \\ z_n^T & 0 \end{bmatrix} \begin{bmatrix} s_n \\ 1 \end{bmatrix} \tag{12.1b}$$

$$s.t. \quad x_{n+1} = A_n x_n + B_n u_n + b_n \tag{12.1c}$$

$$x_0 = \hat{x}_0 \tag{12.1d}$$

$$u_{n,i}^l \leq u_{n,i} \leq u_{n,i}^u \tag{12.1e}$$

$$x_{n,i}^l \leq x_{n,i} \leq x_{n,i}^u \tag{12.1f}$$

$$x_{n,i}^l \leq x_{n,i} + s_{n,i}^l \tag{12.1g}$$

$$x_{n,i} - s_{n,i}^u \leq x_{n,i}^u \tag{12.1h}$$

$$s_{n,i}^l \ , \ s_{n,i}^u \geq 0 \tag{12.1i}$$

$$d_{0,i}^l \leq D_{0,i} u_0 \leq d_{0,i}^u \tag{12.1j}$$

$$d_{n,i}^l \leq C_{n,i} x_n + D_{n,i} u_n \leq d_{n,i}^u \tag{12.1k}$$

$$d_{N,i}^l \leq C_{N,i} x_N \leq d_{N,i}^u \tag{12.1l}$$

where the exponents ^{l} and ^{u} indicate the quantities associated with the lower and upper constraints, and $_i$ is the index of the constraint. The vector s_n is the vector of the slack variables associated with the soft constraints, and it contains the slack variables associated with the lower soft constraint (s_n^l), and the slack variables associated with the upper soft constraint (s_n^u). There are both a quadratic and a linear penalty term on the slack variables of the soft constraints.

In this definition, only the box constraints on states are softened: this keeps the notation somehow lighter, the extension to soft general polytopic constraints being straightforward.

Notice that, as a difference with respect to the unconstrained MPC problem in 7.1 there are no additional equality constraints at the last stage. In case of need, equality constraints can be modeled as box constraints with equal lower and upper bound, and processed directly as inequality constraints.

As an example, the IPM implemented in the HPMPC library is an infeasible-start Mehrotra's predictor-corrector variant, that can handle this case of problems not strictly feasible with respect to the inequality constraints.

12.1.1 Matrix formulation

In order to simplify the notation let us assume that all inputs are hard bounded, and all states are soft bounded.

The linear MPC problem can be reformulated in the matrix form

$$\begin{aligned} \min_{y} \quad & \frac{1}{2}\bar{y}^T\bar{\mathcal{H}}\bar{y} + \bar{g}^T\bar{y} + \gamma \\ s.t. \quad & \bar{\mathcal{A}}\bar{y} = \bar{b} \\ & \bar{\mathcal{C}}\bar{y} \geq \bar{d} \end{aligned} \tag{12.2}$$

where

$$\begin{aligned} \min_{s,u,x} \quad & \frac{1}{2}\begin{bmatrix}\bar{s}\\ \bar{u}\\ \bar{x}\end{bmatrix}^T \begin{bmatrix}\bar{Z} & & \\ & \bar{R} & \bar{S}\\ & \bar{S}^T & \bar{Q}\end{bmatrix}\begin{bmatrix}\bar{s}\\ \bar{u}\\ \bar{x}\end{bmatrix} + \begin{bmatrix}\bar{z}\\ \bar{r}\\ \bar{q}\end{bmatrix}^T\begin{bmatrix}\bar{s}\\ \bar{u}\\ \bar{x}\end{bmatrix} \\ s.t. \quad & \begin{bmatrix}0 & \bar{B} & \bar{A}\end{bmatrix}\begin{bmatrix}\bar{s}\\ \bar{u}\\ \bar{x}\end{bmatrix} = \bar{b} \\ & \begin{bmatrix}\bar{D}_b & \\ I_{2Nx} & \bar{C}_b\\ \bar{D}_g & \bar{C}_g\end{bmatrix}\begin{bmatrix}\bar{s}\\ \bar{u}\\ \bar{x}\end{bmatrix} \geq \begin{bmatrix}\bar{d}_{b,u}\\ \bar{d}_{b,x}\\ \bar{d}_g\end{bmatrix} \end{aligned}$$

where in turn (for $N = 3$) $\bar{u}$, $\bar{x}$, $\bar{A}$, $\bar{B}$, $\bar{b}$, $\bar{Q}$, $\bar{R}$, $\bar{S}$, $\bar{q}$, $\bar{r}$ are defined in 7.3, I_{2Nx}

is the identity matrix of size $2Nn_x$ and

$$\bar{s} = \left[\begin{array}{c} s_1^l \\ s_2^l \\ s_3^l \\ \hline s_1^u \\ s_2^u \\ s_3^u \end{array}\right], \quad \bar{Z} = \left[\begin{array}{ccc|ccc} Z_1^l & & & & & \\ & Z_2^l & & & & \\ & & Z_3^l & & & \\ \hline & & & Z_1^u & & \\ & & & & Z_2^u & \\ & & & & & Z_3^u \end{array}\right],$$

$$\bar{C}_b = \left[\begin{array}{ccc} I_x & & \\ & I_x & \\ & & I_x \\ \hline -I_x & & \\ & -I_x & \\ & & I_x \end{array}\right], \quad d_{b,x} = \left[\begin{array}{c} x_1^l \\ x_2^l \\ x_3^l \\ \hline x_1^u \\ x_2^u \\ x_3^u \end{array}\right], \quad \bar{D}_b = \left[\begin{array}{ccc} I_u & & \\ & I_u & \\ & & I_u \\ \hline -I_u & & \\ & -I_u & \\ & & I_u \end{array}\right], \quad d_{b,u} = \left[\begin{array}{c} u_0^l \\ u_1^l \\ u_2^l \\ \hline u_0^u \\ u_1^u \\ u_2^u \end{array}\right],$$

$$\bar{D}_g = \left[\begin{array}{ccc} D_0 & & \\ & D_1 & \\ & & D_2 \\ \hline -D_0 & & \\ & -D_1 & \\ & & -D_2 \end{array}\right], \quad \bar{C}_g = \left[\begin{array}{ccc} C_1 & & \\ & C_2 & \\ & & C_3 \\ \hline -C_1 & & \\ & -C_2 & \\ & & -C_3 \end{array}\right], \quad d_g = \left[\begin{array}{c} d_0^l \\ d_1^l \\ d_2^l \\ d_3^l \\ \hline d_0^u \\ d_1^u \\ d_2^u \\ d_3^u \end{array}\right]. \tag{12.3}$$

The matrices I_x and I_u are the identity matrices of size n_x and n_u respectively.

The matrices of the inequality constraints are highly structured, and this structure should be exploited in the solvers.

12.1.2 Optimality conditions

The KKT optimality conditions for the constrained MPC problem can be found e.g. in [31].

Introducing the slack variables $\bar{t} = \bar{\mathcal{C}}\bar{y} - \bar{d} \geq 0$, the KKT conditions for (12.2) are

$$\bar{\mathcal{H}}\bar{y} + \bar{g} - \bar{\mathcal{A}}^T\bar{\pi} - \bar{\mathcal{C}}^T\bar{\lambda} = 0 \tag{12.4a}$$
$$\bar{\mathcal{A}}\bar{y} - \bar{b} = 0 \tag{12.4b}$$
$$\bar{\mathcal{C}}\bar{y} - \bar{d} - \bar{t} = 0 \tag{12.4c}$$
$$\bar{\lambda}^T\bar{t} = 0 \tag{12.4d}$$
$$(\bar{\lambda}, \bar{t}) \geq 0 \tag{12.4e}$$

where $\bar{\pi}$ are the Lagrangian multipliers of the equality constraints and $\bar{\lambda}$ are the Lagrangian multipliers of the inequality constraints. Equations (12.4a)-(12.4d) form a system of non-linear equations $f(\bar{y}, \bar{\pi}, \bar{\lambda}, \bar{t}) = 0$ due to the non-linear equation (12.4d). The solution of this system of equations is made harder by the constraint on the sign of Lagrangian multipliers and slack variables in (12.4e).

Chapter 13

Solution of sub-problems in linear MPC and MHE problems

The algorithms for unconstrained MPC and MHE problems can be used as routines in optimization algorithms for constrained and nonlinear MPC and MHE problems.

Since the computationally most expensive part of optimization algorithms for constrained MPC and MHE problems is typically the solution of systems of linear equations in the form of unconstrained MPC and MHE problems, the use of tailored algorithms for these problems can greatly reduce the solution time.

The algorithm presented in this chapter are a far-from-complete selection of optimization algorithms commonly employed in MPC. These algorithms employ the backward Riccati recursion presented in Section 8.1 for the solution of the sub-problems in the form of unconstrained MPC problems. The backward Riccati recursion has been chosen over other solution methods such as the forward Schur complement recursion or the various condensing methods since it provides reasonably good performance over a wide range of problem sizes. Other solution methods can perform better on special cases, such as condensing methods (if n_x is much larger than Nn_u), or the forward Schur complement recursion (if there

are only equality constraints on the last stage and no inequality constraints, since this algorithm can directly handle this case).

Furthermore, partial condensing is briefly investigated as a technique to further decrease the solution time of some optimization algorithms.

13.1 Interior-point methods

Interior-point methods (IPM) are second order optimization methods. The methods in this class, containing also Active-Set (AS) methods [28, 29], have been widely used in optimal control, since they use second order information to converge to the solution in few iterations.

In the case of IPM, the number of iterations is rather unaffected by the problem instance: if well initialized, these methods can typically converge in 8-15 iterations. Each iteration is rather computationally heavy, requiring the factorization and solution of a system of linear equations in the computation of the search direction. The factorization makes use of level 3 BLAS and therefore requires a cubic number of flops in the input and state size. The use of IPM to solve linear MPC problems is not new [70, 62, 26, 75], but it is somehow in contrast with the current trend in linear MPC toward the use of first order methods.

In the MPC framework, Riccati-based IPM have been proposed in [70]. As a solver for the unconstrained MPC problem, the Riccati recursion gives a reasonably good performance for a wide range of problem sizes [31]: in particular, the solution time scales linearly with the horizon length. Therefore, a Riccati-based IPM should reasonably give a good performance for a wide range of problem sizes.

13.1.1 Basics about interior-point methods

IPM have widely been employed in the solution of quadratic programs. IPM arises from the application of the Newton method to the solution of the system of nonlinear equations the KKT conditions (12.4), plus the sign conditions in (12.4e). This section summaries some basics about IPM. A detailed presentation can be found e.g. in [92].

IPM methods often relax the equation (12.4d) as

$$\bar{\lambda}^T \bar{t} = \tau_k \tag{13.1}$$

with the parameter τ_k going to zero as the iteration count k increases. Different IPMs primarily differ in the choice of τ_k.

Given an initial guess $(\bar{y}_0, \bar{\pi}_0, \bar{\lambda}_0, \bar{t}_0)$, at each iteration k primal-dual IPMs iteratively attempt to solve (12.4) by performing steps

$$(\bar{y}_{k+1}, \bar{\pi}_{k+1}, \bar{\lambda}_{k+1}, \bar{t}_{k+1}) = (\bar{y}_k, \bar{\pi}_k, \bar{\lambda}_k, \bar{t}_k) + \alpha(\Delta\bar{y}_{\text{aff}}, \Delta\bar{\pi}_{\text{aff}}, \Delta\bar{\lambda}_{\text{aff}}, \Delta\bar{t}_{\text{aff}})$$

in the direction given by the Newton step, found solving the linear system

$$\nabla f(\bar{y}_k, \bar{\pi}_k, \bar{\lambda}_k, \bar{t}_k)(\Delta\bar{y}_{\text{aff}}, \Delta\bar{\pi}_{\text{aff}}, \Delta\bar{\lambda}_{\text{aff}}, \Delta\bar{t}_{\text{aff}}) = -f(\bar{y}_k, \bar{\pi}_k, \bar{\lambda}_k, \bar{t}_k)$$

that takes the form

$$\begin{bmatrix} \bar{\mathcal{H}} & -\bar{\mathcal{A}}^T & -\bar{\mathcal{C}}^T & \\ \bar{\mathcal{A}} & & & \\ \bar{\mathcal{C}} & & & -I \\ & & \bar{T}_k & \bar{\Lambda}_k \end{bmatrix} \begin{bmatrix} \Delta\bar{y}_{\text{aff}} \\ \Delta\bar{\pi}_{\text{aff}} \\ \Delta\bar{\lambda}_{\text{aff}} \\ \Delta\bar{t}_{\text{aff}} \end{bmatrix} = - \begin{bmatrix} \bar{\mathcal{H}}\bar{y}_k - \bar{\mathcal{A}}^T\bar{\pi}_k - \bar{\mathcal{C}}^T\bar{\lambda}_k + g \\ \bar{\mathcal{A}}\bar{y}_k - \bar{b} \\ \bar{\mathcal{C}}\bar{y}_k - \bar{t}_k - \bar{d} \\ \bar{\Lambda}_k\bar{T}_k e - \tau_k \end{bmatrix} \tag{13.2}$$

where $\bar{\Lambda}_k$ and $\bar{T}_k$ are the diagonal matrices with $\bar{\lambda}_k$ and $\bar{t}_k$ on the diagonal, and e is a vector of ones. Notice that the step length α has to be chosen such that the condition (12.4e) is strictly satisfied at each iteration.

The system of equations (13.2) can be rewritten in the form

$$\begin{bmatrix} \bar{\mathcal{H}} & -\bar{\mathcal{A}}^T & -\bar{\mathcal{C}}^T & \\ \bar{\mathcal{A}} & & & \\ \bar{\mathcal{C}} & & & -I \\ & & \bar{T}_k & \bar{\Lambda}_k \end{bmatrix} \begin{bmatrix} \bar{y}_{\text{aff}} \\ \bar{\pi}_{\text{aff}} \\ \bar{\lambda}_{\text{aff}} \\ \bar{t}_{\text{aff}} \end{bmatrix} = - \begin{bmatrix} \bar{g} \\ -\bar{b} \\ -\bar{d} \\ -\tau_k \end{bmatrix} \tag{13.3}$$

and the search direction can then be computed as

$$(\Delta\bar{y}_{\text{aff}}, \Delta\bar{\pi}_{\text{aff}}, \Delta\bar{\lambda}_{\text{aff}}, \Delta\bar{t}_{\text{aff}}) = (\bar{y}_{\text{aff}}, \bar{\pi}_{\text{aff}}, \bar{\lambda}_{\text{aff}}, \bar{t}_{\text{aff}}) - (\bar{y}_k, \bar{\pi}_k, \bar{\lambda}_k, \bar{t}_k). \tag{13.4}$$

From a computational point of view, this formulation as a lower complexity, since the computation of the right-hand-side in (13.2) requires the computation of the residuals (that has a quadratic cost in n_x and n_u), while in (13.3) and (13.4) the computation of the right hand side and the step given the affine iterate have a linear cost in n_x and n_u. For small-scale systems, the computation of residuals has a cost comparable to the factorization of the KKT matrix of the unconstrained problem.

The system in (13.3) can be rewritten in the form

$$\begin{bmatrix} \bar{\mathcal{H}} + \bar{\mathcal{C}}^T \bar{T}_k^{-1} \bar{\Lambda}_k \bar{\mathcal{C}} & -\bar{\mathcal{A}} \\ -\bar{\mathcal{A}} & \end{bmatrix} \begin{bmatrix} \bar{y}_{\text{aff}} \\ \bar{\pi}_{\text{aff}} \end{bmatrix} = - \begin{bmatrix} \bar{g} - \bar{\mathcal{C}}^T (\bar{\lambda}_k + \bar{T}_k^{-1} \bar{\Lambda}_k \bar{d} + \bar{T}_k^{-1} \tau_k) \\ \bar{b} \end{bmatrix} \quad (13.5)$$

that is the KKT system of an equality constrained QP program, where the cost function is updated to take into account the constraints.

13.1.2 Interior-point methods for the linear MPC problem

The use if IPM in the solution of linear MPC problems has been discussed in [70]. In this section, some of the basics are reported.

In the case of the linear MPC problem in (12.2), this system of equations takes the form

$$\begin{bmatrix} \bar{Z} + \bar{\Gamma}_{x,k} & & \bar{\Gamma}_{x,k} \bar{C}_b & \\ & \bar{R} + \bar{D}_b^T \bar{\Gamma}_{u,k} \bar{D}_b + \bar{D}_g^T \bar{\Gamma}_{g,k} \bar{D}_g & \bar{S} + \bar{D}_g^T \bar{\Gamma}_{g,k} \bar{C}_g & -\bar{B}^T \\ \bar{C}_b^T \bar{\Gamma}_{x,k} & \bar{S}^T + \hat{C}_g^T \bar{\Gamma}_{g,k} \bar{D}_g & \bar{Q} + \bar{C}_b^T \bar{\Gamma}_{x,k} \bar{C}_b + \bar{C}_g^T \Gamma_{g,k} \bar{C}_g & -\bar{A}^T \\ & -\bar{B} & -\bar{A} & \end{bmatrix} \cdot$$

$$\cdot \begin{bmatrix} \bar{s}_{\text{aff}} \\ \bar{u}_{\text{aff}} \\ \bar{x}_{\text{aff}} \\ \bar{\pi}_{\text{aff}} \end{bmatrix} = - \begin{bmatrix} \bar{z} - \bar{\gamma}_{x,k} \\ \bar{r} - \bar{D}_b^T \bar{\gamma}_{u,k} - \bar{D}_g^T \bar{\gamma}_{g,k} \\ \bar{q} - \bar{C}_b^T \bar{\gamma}_{x,k} - \bar{C}_g^T \bar{\gamma}_{g,k} \\ \bar{b} \end{bmatrix}$$

where

$$\begin{aligned} \bar{\Gamma}_{u,k} &= (\bar{T}_u)_k^{-1} (\bar{\Lambda}_u)_k \\ \bar{\Gamma}_{x,k} &= (\bar{T}_x)_k^{-1} (\bar{\Lambda}_x)_k \\ \bar{\Gamma}_{g,k} &= (\bar{T}_g)_k^{-1} (\bar{\Lambda}_g)_k \\ \bar{\gamma}_{u,k} &= (\bar{\lambda}_u)_k + \bar{\Gamma}_{u,k} \bar{d}_u + (\bar{T}_u)_k^{-1} \tau_k \\ \bar{\gamma}_{x,k} &= (\bar{\lambda}_x)_k + \bar{\Gamma}_{x,k} \bar{d}_x + (\bar{T}_x)_k^{-1} \tau_k \\ \bar{\gamma}_{g,k} &= (\bar{\lambda}_g)_k + \bar{\Gamma}_{g,k} \bar{d}_g + (\bar{T}_g)_k^{-1} \tau_k. \end{aligned}$$

Since the diagonal matrix $\bar{Z} + \bar{\Gamma}_{x,k}$ is definite positive at each IPM iteration k, we can use Schur complement to eliminate the $\bar{s}_{\text{aff}}$ variable, obtaining the

system

$$\begin{bmatrix} \bar{R} + \bar{D}_b^T \bar{\Gamma}_{u,k} \bar{D}_b + \bar{D}_g^T \bar{\Gamma}_{g,k} \bar{D}_g & \bar{S} + \bar{D}_g^T \bar{\Gamma}_{g,k} \bar{C}_g & -\bar{B}^T \\ \bar{S}^T + \bar{C}_g^T \bar{\Gamma}_{g,k} \bar{D}_g & \bar{Q} + \bar{C}_b^T \widetilde{\Gamma}_{x,k} \bar{C}_b + \bar{C}_g^T \bar{\Gamma}_{g,k} \bar{C}_g & -\bar{A}^T \\ -\bar{B} & -\bar{A} & \end{bmatrix} \begin{bmatrix} \bar{u}_{\text{aff}} \\ \bar{x}_{\text{aff}} \\ \bar{\pi}_{\text{aff}} \end{bmatrix} = \\ = \begin{bmatrix} \bar{r} - \bar{D}_b^T \bar{\gamma}_{u,k} - \bar{D}_g^T \bar{\gamma}_{g,k} \\ \bar{q} - \bar{C}_b^T \widetilde{\gamma}_{x,k} - \bar{C}_g^T \bar{\gamma}_{g,k} \\ \bar{b} \end{bmatrix} \quad (13.6)$$

where

$$\begin{aligned} \widetilde{\Gamma}_{x,k} &= \bar{\Gamma}_{x,k} + \bar{\Gamma}_{x,k} (\bar{Z} + \bar{\Gamma}_{x,k})^{-1} \bar{\Gamma}_{x,k} \\ \widetilde{\gamma}_{x,k} &= \bar{\gamma}_{x,k} + \bar{\Gamma}_{x,k} (\bar{Z} + \bar{\Gamma}_{x,k})^{-1} (\bar{z} - \bar{\gamma}_{x,k}) \end{aligned} \quad (13.7)$$

The take-home idea about the use of IPMs for MPC problems is that system (13.6) is the KKT system of a linear-quadratic control problem (7.1). Therefore the search direction at each IPM iteration can be found by means of a solver for problem (7.1).

Note that the constraints matrices $\bar{C}_b$ and $\bar{D}_b$ of the box constraints are highly structured and very sparse: this can be exploited to compute the update to the $\bar{R}$, $\bar{Q}$ matrices and the $\bar{r}$, $\bar{q}$ vectors at a cost $\mathcal{O}(Nn_b)$, linear in the number box constraints. In case of general polytopic constraints, the update to the matrices $\bar{R}$, $\bar{S}$ and $\bar{Q}$ can be computed at a cost $\mathcal{O}(Nn_g(n_u + n_x)^2)$, while the update to the vectors $\bar{r}$ and $\bar{q}$ can be computed at a cost $\mathcal{O}(Nn_g(n_u + n_x))$.

The only extra cost due to the introduction of soft constraints is the computation of the matrices $\widetilde{\Gamma}_{x,k}$ and $\widetilde{\gamma}_{x,k}$ in (13.7), requiring $\mathcal{O}(Nn_s)$ flops, linear in the number of soft constraints.

13.1.3 Interior-point methods implementation choices

The IPM considered in this thesis is the infeasible start Mehrotra's predictor-corrector IPM [64], proposed in the MPC context e.g. in [70, 26].

The computation of the search direction in (13.6) is the key step in IPMs, typically requiring most of the computation time. In the implemented Riccati-based IPM, the prediction direction is computed solving a system in the form (7.1) by means of the backward Riccati recursion presented in details in Section 8.1: the cost of this step is therefore cubic in the number of optimization variables and linear in the horizon length. The correction direction is computed reusing the KKT matrix factorized while computing the prediction direction: the cost of this step is quadratic in the number of optimization variables and linear in

the horizon length. All other operations in the IPM have cost linear in both the number of optimization variables and the horizon length.

Therefore, for large-scale problems the cost at each IPM iteration is dominated by the cost to compute the prediction search direction. However, for the small-scale problems typical of MPC, quadratic and linear cost terms accounts for a considerable fraction of the IPM iteration cost. Therefore, particular care is used in their implementation, by reducing memory movements or using vector instructions when supported by hardware. The use of faster single-precision division instructions can speed-up the computation of the step length α, if exact line search is employed in place of backtracking.

Generally, IPM can converge in few iterations to a solution, if well initialized. Warm starting in not particularly beneficial, and on the contrary it can considerably increases the number of iterations in case of large disturbances. Therefore, as a general rule, cold stating is preferred in case of IPM. Even if infeasible-start IPM can converge also if the initial guess is infeasible, a good choice of the initial guess can reduce the number of iterations. In particular, a good initial guess lies away from the constraints, close to the central path [92]. Furthermore, the slack variables and the associated Lagrangian multipliers should be chosen such that the initial value of the duality measure μ_0 is not too small compared to the largest Lagrangian multiplier at the solution. Empirically, if the MPC problem is well scaled, a good choice for μ_0 is the largest absolute value of the elements in the cost function matrices. In particular, this is found to work well also in the soft-constrained case, where typically the penalty on the slack variables of the soft constraints is set to very large values. With this choice, the number of iterations in the soft-constrained case is only slightly larger than in the box-constrained case.

As a final note, the backward Riccati recursion routines presented in Section 8.1 can deal with input and state vector sizes varying at each stage, and the IPM is implemented in the same fashion. This allows the use of partial condensing as a preparation step before the IPM for all possible values of the horizon N_p of the partially condensed MPC problem.

13.1.4 Partial condensing for linear MPC problems

This section investigates the effect of partial condensing in the performance of solvers for the linear MPC problem (12.1). The presentation is limited to the case of bounds in the original MPC problem, the extension to general polytopic constraints being straightforward.

Partial condensing can be employed to reformulate an MPC problem into another one with shorter horizon length and larger input vector size. This is analogue to the case of the unconstrained MPC problem (7.1), with the difference that in the case of problem (12.1) also the constraints need to be partially condensed.

13.1.4.1 Condensing of bounds on states

The bounds on states that are condensed become general polytopic constraints on the inputs and remaining states. Two cases can be distinguished: all states are condensed (called in the following 'MPC case', since it is analogue to the condensing of a MPC problem), or all states are condensed beside the first one (called in the following 'MHE case', since it is analogue to the condensing of a MHE problem).

Let us consider a state constraint in the form

$$\bar{x}^l \leq \bar{C}\bar{x} \leq \bar{x}^u \tag{13.8}$$

where the matrix $\bar{C}$ describes the constraint shape. In case of bounds on all state components, the matrix $\bar{C}$ is the identity: this case will be assumed in the following, the extension to the general case being straightforward.

In the remaining of Section 13.1.4.1 it is assumed that the MPC and MHE problems to be condensed have horizon length N, state vector size n_x and input vector size n_u, and therefore these quantities do not refer to the size of the MPC problem to be partially condensed.

MPC case Inserting the expression for $\bar{x}$ in (9.1), equation (13.8) becomes

$$\bar{x}^l - \Gamma_{x,b} \leq \Gamma_u \bar{u} \leq \bar{x}^u - \Gamma_{x,b}$$

that can be considered as a general polytopic constraint on the inputs. The matrix Γ_u has size $Nn_x \times Nn_u$, and therefore if it is processed as the matrix of the general polytopic constraints in an IPM (i.e. $\bar{D}_g = \Gamma_u$), the update of the condensed Hessian matrix H_R at each iteration of the IPM

$$H_R + \bar{D}_g^T \bar{\Gamma}_{g,k} \bar{D}_g$$

(where the matrix $\Gamma_{g,k}$ is diagonal) can be computed in about $N^3 n_x n_u^2$ flops.

The IPM could be modified to exploit the structure of the Γ_u matrix. In this case, the update can be computed in $\frac{1}{3}N^3 n_x n_u^2$ flops by exploiting the fact that

Γ_u is block lower triangular, similarly to the computation of the term $\Gamma_u^T \bar{Q} \Gamma_u$ in Algorithm 9. Alternatively, the fact that $\Gamma_u = \bar{A}^{-1}\bar{B}$ can be exploited similarly to Algorithms 10 and 11.

If the condensing algorithm is employed as a routine in partial condensing, the bound on the last state is not condensed, and therefore the matrix Γ_u has size $(N-1)n_x \times Nn_u$. This slightly reduces the computational cost.

MHE case Inserting the expression for $\bar{x}$ in (9.22), equation (13.8) becomes

$$\bar{x}^l - \Gamma_b \leq \Gamma_v \bar{v} \leq \bar{x}^u - \Gamma_b$$

that can be considered as a general polytopic constraint on the inputs and the first state. The matrix Γ_v has size $Nn_x \times (n_x + Nn_u)$, and therefore if it is processed as the matrix of the general polytopic constraints in an IPM (i.e. $\bar{D}_g = \Gamma_v$), the update of the $\mathcal{H}_R$ matrix at each iteration of the IPM

$$\mathcal{H}_R + \bar{D}_g^T \bar{\Gamma}_{g,k} \bar{D}_g$$

(where the matrix $\Gamma_{g,k}$ is diagonal) can computed in about $Nn_x^3 + 2N^2 n_x^2 n_u + N^3 n_x n_u^2$ flops.

Also in the MHE case the IPM could be modified to exploit the structure of the Γ_u matrix. In this case, the update can be computed in $Nn_x^3 + N^2 n_x^2 n_u + \frac{1}{3}N^3 n_x n_u^2$ flops by exploiting the fact that Γ_u is block lower triangular, similarly to the computation of the term $\Gamma_u^T \bar{Q} \Gamma_u$ in Algorithm 20. Alternatively, the fact that $\Gamma_u = \bar{A}^{-1}\bar{B}$ can be exploited similarly to Algorithms 21 and 22.

If the condensing algorithm is employed as a routine in partial condensing, the bound on the last state is not condensed, and therefore the matrix Γ_u has size $(N-1)n_x \times (n_x + Nn_u)$. This slightly reduces the computational cost.

13.1.5 Comparison of solvers for linear MPC problems

This section presents the results of some test for the IPM solver part of the HPMPC library. The predictor and corrector search directions are computed using the Riccati solvers in Algorithm 1 and Algorithm 3, tested in Section 8.3. The IPM solver can be called through two interfaces: a low-level interface, and an high-level interface that is a wrapper around the low-level one (see Figure 1.1).

The low-level interface exposes all features of the IPM solver, and gives the best performance. It requires all matrices to be stored using the panel-major matrix

format in Figure 3.1: this avoids the cost of converting them from row-major or column-major orders on-line. The data matrices are passed to the solver as `double**` type, and this can be exploited to reduce memory usage in case of time-invariant systems or cost functions. All data matrices and vectors are assumed to be properly aligned in memory to SIMD or cache line boundaries: these, together with the panel width b_s, are architecture-dependent quantities.

The high-level interface is a wrapper hiding all architecture-dependent details and converting data from row-major or column-major orders into the panel-major format. Therefore it adds the overhead of this on-line conversion to the solver execution time. Furthermore, the ability to reduce memory usage in case of time-invariant data is lost. In fact, the data matrices are passed to this interface as `double*` type, i.e. as a sequences of matrices, with matrices for the different stages following each other (i.e. as produced using the `repmat` function in Matlab). This interface is designed to simplify the integration of the solver into existing software, at the cost of some performance penalty.

Unless differently stated, all tests in the remainder of the chapter are performed on the laptop equipped with the Intel Core i7 4800MQ already considered in this thesis. The linear algebra is given by the AVX2 target in HPMPC.

13.1.5.1 Box-constrained linear MPC problems

In this section the performance of the proposed IPM solver is compared against the successful IPM solver FORCES [26] and the IPM solver in FORCES_Pro [8] in the solution of a box-constrained linear MPC problem.

The test problem is the linear mass-spring system often used as benchmark for linear solvers in MPC [89]. Namely, the tests performed in [26] (where the FORCES solver is showed to outperform by a wide margin the solvers CVXGEN [62], CPLEX [7] and MA57 [27] in this benchmark) are repeated. The only exception is the largest problem: we had to decrease the horizon length from 30 to 15 in case of the FORCES solver and to 10 in case of the FORCES_Pro solver (for larger test problems the server running the code generation returns a timeout error), and therefore the value obtained for $N = 30$ is an estimation based on the value obtained for $N = 10$ and the assumption that the solution time is linear.

The dynamic system to be controlled is the undamped linear mass-spring system, consisting of a chain of M masses connected each other with springs. The system is modeled using 2 states per mass, one representing the deviation of the mass from the rest position and one representing the mass velocity (the total

number of states is $n_x = 2M$). The inputs are n_u forces acting on the first n_u masses. The continuous-time system is discretized with a sampling time of 0.5 seconds: therefore the discrete-time matrices A_n and B_n in (12.1) are dense. Forces acts on all but the last mass, so $n_u = M - 1$. There are $n_b = n_u + n_x$ two-sided box constraints: inputs must satisfy the bounds $|u| \leq 0.5$, while states must satisfy the bounds $|x| \leq 4$. The cost-function matrices are $R = 2I$, $Q = P = I$, $S = 0$, $r = 0$, $q = p = 0$. Notice that the test problem is linear time-invariant.

Both the low- and high-level interfaces of the IPM in HPMPC are tested against each other and against FORCES and FORCES_Pro. The results of the test are in Table 13.1, where the number of IPM iterations is fixed to 10 for all solvers.

In the comparison of the low- and high-level HPMPC interfaces, for all problems the high-level interface is about 10% slower than the low-level one, showing that the on-line cost for packing matrices is well amortized over the 10 IPM iterations.

The comparison of the IPM solver in HPMPC with FORCES or FORCES_Pro clearly shows the performance gains of the proposed implementation approach. The performance of FORCES and FORCES_Pro is rather similar, with the latter showing small performance improvement. For this benchmark problem, FORCES (and presumably FORCES_Pro) has a slightly lower cost in terms of flops with respect to the Riccati-based IPM in HPMPC. However, the IPM in HPMPC is from about 2 (smallest problem) to more than 10 (largest problem) times faster than FORCES_Pro, with the performance gap increasing with the problem size. Also the high-level interface in HPMPC is considerably faster that FORCES_Pro, proving that in an MPC framework it makes sense to invest time in arranging data in a way optimal for the linear-algebra routines, even if this has to be done on-line.

13.1.5.2 Soft-constrained linear MPC problems

This section tests the performance of the IPM solver in HPMPC in the solution of a soft-constrained linear MPC problem. Two test are performed: the first handles the soft constraints using the specialized interface, while as a reference the second handles them as general polytopic constraints.

The tests are the same performed in Section 13.1.5.1, with the difference that the bounds are tightened in order to make the problem unfeasible if bounds are considered as hard. Namely, the bounds on inputs are still $|u| \leq 0.5$, but the bounds on states are now $|x| \leq 1$. Furthermore, the cost function matrices are now $Q = P = 0$, $R = 2I$, $S = 0$, $q = p = 0$ and $r = 0$: the expected behavior

Table 13.1: Comparison of solvers for the box-constrained linear MPC problem: low- and high-level interfaces for the IPM in HPMPC, FORCES IPM and FORCES_Pro IPM. Run times are presented in seconds. For each problem size and solver, the number of IPM iterations is fixed to 10. Blue: estimated values based on the results for $N = 15$ (FORCES) and $N = 10$ (FORCES_Pro) and the assumption of linear solution time in N.

M	n_x	n_u	n_b	N	HPMPC low-level	HPMPC high-level	FORCES	FORCES Pro
2	4	1	5	10	$5.39 \cdot 10^{-5}$	$6.31 \cdot 10^{-5}$	$1.1 \cdot 10^{-4}$	$1.0 \cdot 10^{-4}$
4	8	3	11	10	$9.05 \cdot 10^{-5}$	$1.04 \cdot 10^{-4}$	$3.4 \cdot 10^{-4}$	$3.1 \cdot 10^{-4}$
6	12	5	17	30	$5.07 \cdot 10^{-4}$	$5.74 \cdot 10^{-4}$	$2.11 \cdot 10^{-3}$	$1.84 \cdot 10^{-3}$
11	22	10	32	10	$3.94 \cdot 10^{-4}$	$4.60 \cdot 10^{-4}$	$3.96 \cdot 10^{-3}$	$3.29 \cdot 10^{-3}$
15	30	14	44	10	$7.03 \cdot 10^{-4}$	$8.17 \cdot 10^{-4}$	$9.47 \cdot 10^{-3}$	$7.49 \cdot 10^{-3}$
30	60	29	89	30	$1.10 \cdot 10^{-2}$	$1.26 \cdot 10^{-2}$	$1.67 \cdot 10^{-1}$	$1.25 \cdot 10^{-1}$

for the controller is to let the masses to freely oscillate around the rest position, while ensuring that displacements and velocities of masses satisfy the bound $|x| \leq 1$.

When the solver tailored to handle soft-constraints is considered, the problem has $n_b = n_u$ two-sided box hard constraints on inputs, and $n_s = n_x$ two-sided box soft constraints on states. The slack variables s in (12.1) are not explicitly considered as dynamical system variables, but rather are internally handled by the solver as shown in Section 13.1.2. This keeps the size of the dynamic system as small as in the hard-constrained case.

When the generic IPM solver in HPMPC is considered, the slack variables s have to be considered as extra input variables: in the current test problem, this meas that the new input size is $\tilde{n}_u = n_u + 2n_x$. In order to keep the number of constraints as small as possible, the two one-sided constraints (12.1g) and (12.1h) are rewritten as the single two-sided constraint

$$x^l_{n,i} \leq x_{n,i} + s^l_{n,i} - s^u_{n,i} \leq x^u_{n,i}.$$

The condition on the sign of the slack variables (12.1i) and the penalties $Z_l > 0$ or $z_l > 0$ ensure the equivalence of the two formulations at the optimum. Therefore there are $\tilde{n}_g = n_x$ two-sided general constraints. The $2n_x$ one-sided constraint on the sign of the slack variables (12.1i) are handled as the lower bound of $2n_x$ two-sided box constraints, where the upper bound is arbitrary fixed to a value large enough. In total, there are $\tilde{n}_b = n_u + 2n_x$ two-sided box constraints and $\tilde{n}_g = n_x$ general polytopic constraints per stage. Compared to

Table 13.2: Comparison of solvers for the soft-constrained linear MPC problem: tailored IPM solver (soft-box) and generic IPM solver for (hard) general polytopic constraints (hard-gen). Run times are presented in seconds. For each problem size and solver, the number of IP iterations is fixed to 10.

M	n_x	n_u	N	n_b	n_s	HPMPC soft-box	$\tilde{n}_u$	$\tilde{n}_b$	$\tilde{n}_g$	HPMPC hard-gen
2	4	1	10	1	4	$7.0 \cdot 10^{-5}$	9	9	4	$1.3 \cdot 10^{-4}$
4	8	3	10	3	8	$1.2 \cdot 10^{-4}$	19	19	8	$2.9 \cdot 10^{-4}$
6	12	5	30	5	12	$6.4 \cdot 10^{-4}$	29	29	12	$1.9 \cdot 10^{-3}$
11	22	10	10	10	22	$4.6 \cdot 10^{-4}$	52	52	22	$2.2 \cdot 10^{-3}$
15	30	14	10	14	30	$8.7 \cdot 10^{-4}$	74	74	30	$4.6 \cdot 10^{-3}$
30	60	29	30	29	60	$1.2 \cdot 10^{-2}$	149	149	60	$9.3 \cdot 10^{-2}$

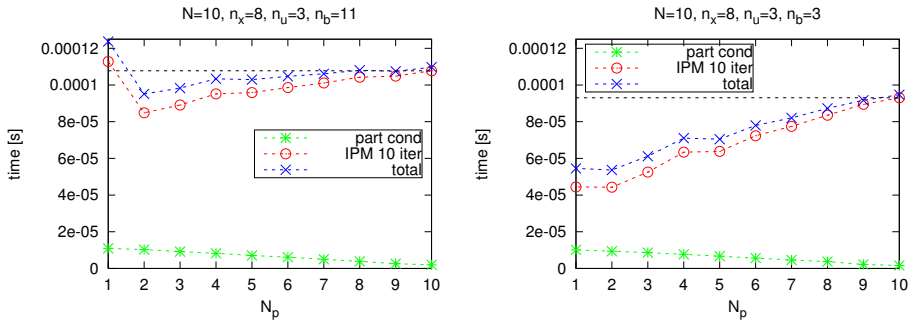
the (hard) box-constrained test problem in Table 13.1, the solver tailored to the soft-constrained linear MPC problem is about 20% slower for the smaller problems, down to about 10% slower for the largest ones. On the other hand, the general IPM solver applied to the soft-constrained linear MPC problem is from 2x slower (smaller problem) to about 8x slower than the tailored solver, due to the increased size of the dynamical system.

13.1.5.3 Partial condensing

This section contains the results of some numerical investigation on the use of partial condensing to speed up the solution of linear MPC problems.

More precisely, partial condensing is employed as a preparation step before the call to the IPM solver, with the aim of changing the size of the linear MPC problem. The IPM has not been tailored to take advantage of the special structure of the Γ_u matrix, that therefore is processed as the constraint matrix of general polytopic constraints.

The test problem is the mass-spring problem with $N = 10$, $n_x = 8$ and $n_u = 3$. Two series of tests are performed: in one all inputs and states are bounded (and therefore $n_b = 11$), in the other one only inputs are bounded (and therefore $n_b = 3$). These are the two limit cases, with the former being the worst case (all state bounds beside the last one are turned into general polytopic constraints), while the latter being the best case (there are no general polytopic constraints). The performance of all other cases is in the middle between the two limit cases.



(a) All states bounded.

(b) No state bounded.

Figure 13.1: Solution time for the mass-spring MPC problem with $N = 10$, $n_x = 8$, $n_u = 3$ and either bounds on all inputs and states ($n_b = 11$) or input bounds only ($n_b = 3$). A combination of partial condensing and a predictor-corrector IPM (fixed to 10 iterations) is employed for different values of the horizon length N_p for the partially condensed MPC problem. Test performed on an Intel Core i5 3520M processor, `gcc` compiler, AVX target ISA in HPMPC.

Figure 13.1 contains the results of some numerical tests, for values of the horizon length of the partially condensed MPC problem $N_p \in [1, 10]$. In the partial condensing, the linear MPC problem is assumed to be time-variant, and therefore the matrices are recomputed at each stage of the partially condensed MPC problem: this is the worst case, and it is the expected behavior in solving the linear MPC sub-problems in NMPC.

The cost of partial condensing is well amortized over the 10 IPM iterations, and accounts for a small fraction of the total computational time. The use of partial condensing is particularly advantageous in case there are input constraints only. In this case, for $N_p = 2$ there is a reduction in the computational time of about 40%. On the other hand, in case all states are bounded, the reduction in computational time is of about the 10%, since the update of the cost function at each iteration of the IPM makes up for most of the savings.

13.2 Alternating direction method of multipliers

The Alternating Direction Method of Multipliers (ADMM) is a first order optimization method belonging to the class of splitting method. ADMM has been firstly proposed in the field of optimal control in [68].

First order optimization methods (as gradient methods [74, 73] and splitting methods [68, 82]) are generally straightforward to implement and can easily exploit sparsity and special problem structures. They perform many but cheap iterations, where the cost-per-iteration is quadratic in the input and state size (requiring level 2 BLAS operations as e.g a matrix-vector multiplication or solution of a system of linear equation whose matrix is factorized off-line). In general, the number of iterations (and therefore the solution time) can vary significantly with the number of active constraints and the problem conditioning. Generally speaking, first order methods can find quickly an approximate solution, but require a large number of iterations to improve the accuracy of the solution.

13.2.1 Notation and basics about ADMM

This section briefly reports some basics about ADMM. A more detailed presentation can be found in [68]. The notation follows the one in that reference.

The Alternating Direction Method of Multipliers (ADMM), also known as Douglas-Rachford (D-R) splitting, is an operator splitting method. It breaks the problem into two parts, a quadratic optimal control problem and a set of single period optimization problems. The splitting is generally not unique. The solution is found iterating over these two steps.

In the following, only the case of the linear MPC problem is considered. In this framework, $\phi(\bar{x}, \bar{u})$ indicates the (convex) cost function of the quadratic cost function of the linear MPC problem,

$$\phi(\bar{x}, \bar{u}) = \frac{1}{2} \sum_{n=0}^{N-1} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix}^T \begin{bmatrix} R_n & S_n & r_n \\ S_n^T & Q_n & q_n \\ r_n^T & q_n^T & 0 \end{bmatrix} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} x_N \\ 1 \end{bmatrix}^T \begin{bmatrix} Q_N & q_N \\ q_N^T & 0 \end{bmatrix} \begin{bmatrix} x_N \\ 1 \end{bmatrix}.$$

The set $\mathcal{C}$ indicates the close convex feasible set with respect to the inequality constraints. Therefore, $\psi(\bar{x}, \bar{u})$ indicates the (convex) non-quadratic cost function used to describe the inequality constraints, as the indicator function $I_{\mathcal{C}}$ of

the set $\mathcal{C}$,

$$\psi(\bar{x},\bar{u}) = I_{\mathcal{C}}(\bar{x},\bar{u}) = \begin{cases} 0, & (\bar{x},\bar{u}) \in \mathcal{C} \\ \infty, & (\bar{x},\bar{u}) \notin \mathcal{C} \end{cases}$$

Similarly, $\mathcal{D}$ indicates the set of the feasible pairs $(\bar{x},\bar{u})$ with respect to the equality constraints (system dynamic equations)

$$\mathcal{D} = \{(\bar{x},\bar{u}) | x_0 = \hat{x}_0, \quad x_{n+1} = A_n x_n + B_n u_n + b_n, \quad n = 0,1,\ldots,N-1\}$$

and $I_{\mathcal{D}}$ indicates the indicator function of $\mathcal{D}$,

$$I_{\mathcal{D}}(\bar{x},\bar{u}) = \begin{cases} 0, & (\bar{x},\bar{u}) \in \mathcal{D} \\ \infty, & (\bar{x},\bar{u}) \notin \mathcal{D} \end{cases}$$

With this notation, the linear MPC problem can be rewritten as

$$\min_{(\bar{x},\bar{u})} \quad I_{\mathcal{D}}(\bar{x},\bar{u}) + \phi(\bar{x},\bar{u}) + \psi(\bar{x},\bar{u}).$$

An algorithm for ADMM is presented in [68]: starting from any initial guess $(\tilde{x}^0,\tilde{u}^0)$ and $(\bar{z}^0,\bar{y}^0)$, for each iteration k the next iterate is obtained from the recursion

$$(\bar{x}^{k+1},\bar{u}^{k+1}) = \arg\min_{(\bar{x},\bar{u})} \left(I_{\mathcal{D}}(\bar{x},\bar{u}) + \phi(\bar{x},\bar{u}) + \tfrac{\rho}{2}||(\bar{x},\bar{u}) - (\tilde{x}^k,\tilde{u}^k) - (\bar{z}^k,\bar{y}^k)||_2^2\right) \tag{13.9}$$

$$(\tilde{x}^{k+1},\tilde{u}^{k+1}) = \arg\min_{(\tilde{x},\tilde{u})} \left(\psi(\tilde{x},\tilde{u}) + \tfrac{\rho}{2}||(\bar{x}^{k+1},\bar{u}^{k+1}) - (\tilde{x},\tilde{u}) - (\bar{z}^k,\bar{y}^k)||_2^2\right) \tag{13.10}$$

$$(\bar{z}^{k+1},\bar{y}^{k+1}) = (\bar{z}^k,\bar{y}^k) + (\tilde{x}^{k+1},\tilde{u}^{k+1}) - (\bar{x}^{k+1},\bar{u}^{k+1}) \tag{13.11}$$

where $\rho > 0$ is an algorithm parameter.

13.2.2 Box constraints

The case of box constraints is a particularly favorable case for ADMM. In case of box constrained MPC problems, the step in (13.9) requires the solution of an unconstrained problem formally identical to the unconstrained MPC problem (7.1). In fact, defined $(\bar{p},\bar{t}) = -(\tilde{x}^k,\tilde{u}^k) - (\bar{z}^k,\bar{y}^k)$, the penalty on the two norm can be written as

$$\frac{\rho}{2}||(\bar{x},\bar{u}) + (\bar{p},\bar{t})||_2^2 = \frac{1}{2}\sum_{n=0}^{N-1} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix}^T \begin{bmatrix} \rho I & 0 & \rho t_n \\ 0 & \rho I & \rho p_n \\ \rho t_n^T & \rho p_n^T & * \end{bmatrix} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix}$$

that is an update of the cost function. Notice that the quadratic part of the cost function does not depend on $(\bar{p}, \bar{t})$, and therefore in the KKT system (in matrix notation)

$$\begin{bmatrix} \bar{R}+\rho I & \bar{S} & -\bar{B}^T \\ \bar{S}^T & \bar{Q}+\rho I & -\bar{A}^T \\ \bar{B} & \bar{A} & 0 \end{bmatrix} \begin{bmatrix} \bar{u} \\ \bar{x} \\ \bar{\pi} \end{bmatrix} = - \begin{bmatrix} \bar{r}+\rho \bar{t} \\ \bar{q}+\rho \bar{p} \\ \bar{b} \end{bmatrix} \tag{13.12}$$

the KKT matrix can be factorized only once (off-line or at the first iteration), and reused at all subsequent iterations. The cost of this step is thus quadratic in the state and input dimension. The solution of this step, $(\bar{x}^{k+1}, \bar{u}^{k+1})$ is primal feasible, but in general dual infeasible.

Step in (13.10) requires the solution of a QP that does not have equality constraints (there are no dynamic equations) but only box constraints. Defined $(\bar{p}, \bar{t}) = (\bar{x}^{k+1}, \bar{u}^{k+1}) - (\bar{z}^k, \bar{y}^k)$, the direction of the negative gradient of this QP is simply $(\bar{h}, \bar{g}) = -\rho^{-1}(-\rho(\bar{p}, \bar{t})) = (\bar{p}, \bar{t})$. Since the box constraints are completely separable, the solution of this step is computed by clipping $(\bar{p}, \bar{t})$ to the value of the box constraints,

$$(\tilde{x}^{k+1}, \tilde{u}^{k+1}) = \min\big(\max\big((\bar{p}, \bar{t}), (\bar{x}^l, \bar{u}^l)\big), (\bar{x}^u, \bar{u}^u)\big)$$

This step can be computed in time linear in the state and input size. The solution of this step, $(\tilde{x}^{k+1}, \tilde{u}^{k+1})$ is dual feasible, but in general primal infeasible.

Step in (13.11) is simply the consensus step, and it can clearly be computed in time linear in the state and input size.

13.2.3 Soft constraints

The case of soft box constraints is also a rather favorable case for ADMM, at least regarding the computational cost per iteration, and disregarding the possible increase in the number of iterations required to converge.

Namely, the splitting of dynamic and constraints into two separate steps makes it easy to adapt the ADMM scheme to the case of soft constraints on the states. In this case, also the slack variables s of the soft constraints are optimization variables.

In step (13.9), s enter in the cost function formulation, but do not enter in the dynamic. If we define $(\bar{l}, \bar{p}, \bar{t}) = -(\tilde{s}^k, \tilde{x}^k, \tilde{u}^k) - (\bar{w}^k, \bar{z}^k, \bar{y}^k)$, this means that the

new KKT system to be solved at step (13.9)

$$\left[\begin{array}{c|ccc} \bar{Z}+\rho I & 0 & 0 & 0 \\ \hline 0 & R+\rho I & S & -\bar{B}^T \\ 0 & \bar{S}^T & \bar{Q}+\rho I & -\bar{A}^T \\ 0 & \bar{B} & \bar{A} & 0 \end{array}\right] \left[\begin{array}{c} \bar{s} \\ \bar{u} \\ \bar{x} \\ \bar{\pi} \end{array}\right] = - \left[\begin{array}{c} \bar{z}+\rho\bar{w} \\ \bar{r}+\rho\bar{t} \\ \bar{q}+\rho\bar{p} \\ \bar{b} \end{array}\right] \tag{13.13}$$

is block diagonal, since s is completely separated from the other variables. Since the matrix $Z+\rho I$ is diagonal and and constant over the ADMM iterations (and therefore it can be inverted off-line or at the first iteration), and the bottom-right block can be solved exactly as in (13.12), the cost of this step is almost identical to the case with box-constraints.

The computation of step (13.10) is more interesting. The inputs $\bar{u}$ are box constrained, and therefore treated as in the case of box-constrained MPC. The states are soft constrained, but the constraints are completely separable over both the control horizon N and the state components. Therefore, defined $(\bar{l},\bar{p},\bar{t}) = (\bar{s}^{k+1},\bar{x}^{k+1},\bar{u}^{k+1}) - (\bar{w}^k,\bar{z}^k,\bar{y}^k)$, for each stage $n \in \{1,\dots,N\}$ and for each component $i \in \{0,\dots,n_x-1\}$, the variables $\tilde{x}_{n,i}^{k+1}$, $\tilde{s}_{n,i}^{u,k+1}$ and $\tilde{s}_{n,i}^{l,k+1}$ are computed by solving the optimization problem (dropping the iteration index $^{k+1}$ and using the index $_i$ instead of $_{n,i}$ in the equations for clearness of presentation)

$$\begin{array}{ll} \min\limits_{\tilde{x}_i,\tilde{s}_i^u,\tilde{s}_i^l} & \frac{1}{2}\tilde{x}_i^2 - p_i\tilde{x}_i + \frac{1}{2}(\tilde{s}_i^u)^2 - l_i^u\tilde{s}_i^u + \frac{1}{2}(\tilde{s}_i^l)^2 - l_i^l\tilde{s}_i^l \\ s.t. & \tilde{x}_i - \tilde{s}_i^u \le x_i^u \\ & \tilde{s}_i^u \ge 0 \\ & x_i^l \le \tilde{x}_i + \tilde{s}_i^l \\ & \tilde{s}_i^l \ge 0 \end{array} \tag{13.14}$$

This optimization problem can be solved analytically. In fact, since the upper and the lower soft constraints can not be violated at the same time, there are only three possible cases:

- both the upper and the lower soft constraints are inactive. In this case, both $\tilde{s}_{n,i}^{u,k+1} = 0$ and $\tilde{s}_{n,i}^{l,k+1} = 0$, and the minimization problem (13.14) reduces to

  $$\min_{\tilde{x}_i} \quad \frac{1}{2}\tilde{x}_i^2 - p_i\tilde{x}_i$$

  that has the solution (with all indexes)

  $$\tilde{x}_{n,i}^{k+1} = p_{n,i}$$

  as in the box-constrained case when the constraints are inactive.

- the bottom soft constraint is inactive, but the upper one is active. This means that $\tilde{s}_{n,i}^{l,k+1} = 0$ and $\tilde{x}_{n,i}^{k+1} - \tilde{s}_{n,i}^{u,k+1} = x_{n,i}^{u}$. Inserting these values in the cost function expression in (13.14) gives the minimization problem

$$\min_{\tilde{s}_i^u} \quad (\tilde{s}_i^u)^2 - (-x_i^u + p_i + l_i^u)\tilde{s}_i^u$$

  giving the analytic solution (with all indexes)

$$\tilde{s}_{n,i}^{u,k+1} = \tfrac{1}{2}(-x_{n,i}^{u} + p_{n,i} + l_{n,i}^{u}) \geq 0.$$

- the upper soft constraint is inactive, but the bottom one is active. This means that $\tilde{s}_{n,i}^{u,k+1} = 0$ and $x_{n,i}^{l} = \tilde{x}_{n,i}^{k+1} + \tilde{s}_{n,i}^{l,k+1}$. Inserting these values in the cost function expression in (13.14) gives the minimization problem

$$\min_{\tilde{s}_i^l} \quad (\tilde{s}_i^l)^2 - (x_i^l - p_i + l_i^l)\tilde{s}_i^l$$

  giving the analytic solution (with all indexes)

$$\tilde{s}_{n,i}^{l,k+1} = \tfrac{1}{2}(x_{n,i}^{l} - p_{n,i} + l_{n,i}^{l}) \geq 0.$$

Since for each stage n and index i the computation requires a constant number of flops, the step in (13.10) can be computed in time linear in state and input dimension also in the case of soft-constraints.

Step in (13.11) is again simply the consensus step, and it can clearly be computed in time linear in the state and input size. Notice that this time also the consensus variables w^{k+1} associated with the slack variables s are updated in this step.

13.2.4 ADMM implementation choices

The key step in the ADMM algorithm is the solution of the KKT system at each iteration. Since the KKT matrix does not change at different iterations, it can be factorized once and reused several times (or even factorized off-line in case of time-invariant problems), well amortizing the factorization cost.

The factorization and the solution are performed using the backward Riccati Algorithms 2 and 3 respectively. Riccati-based ADMM implementations can be found in [17, 76]. In the solution Algorithm 3, it is possible to save many flops and use much less memory if the matrices $L_{n,22}$ are not employed. This is the case if the Lagrangian multipliers λ are not computed, and if the quantity $P_{n+1}b_n$ is precomputed during the factorization step. A further advantage of the use of less memory is that the solution routine attains a better computational performance, since the computational performance steadily decreases as soon as the MPC problem memory footprint exceeds cache levels [37].

Table 13.3: Numerical test for the ADMM solver implemented using routines in HPMPC in the solution of the box-constrained linear MPC problem. Run times are presented in seconds. For each problem size and solver, the number of ADMM iterations is fixed to 50.

M	n_x	n_u	n_b	N	HPMPC ADMM
2	4	1	5	10	$7.02 \cdot 10^{-5}$
4	8	3	11	10	$1.06 \cdot 10^{-4}$
6	12	5	17	30	$5.13 \cdot 10^{-4}$
11	22	10	32	10	$3.08 \cdot 10^{-4}$
15	30	14	44	10	$4.65 \cdot 10^{-4}$
30	60	29	89	30	$4.73 \cdot 10^{-3}$

13.2.5 Numerical results for the linear MPC problem

In this section, the numerical tests for the box-constrained and soft-constrained linear MPC problems (employed in the testing of the IPM solver in Section 13.1.5) are repeated for the ADMM solver.

13.2.5.1 Box-constrained linear MPC problems

This section contains the results on the use of the ADMM in the solution of box-constrained MPC problems. The results are in table 13.3, where the size of the problem is the same as in the IPM case in table 13.1. The factorization is performed off-line, and the number of ADMM iterations is fixed to 50.

Generally speaking, the ADMM is slightly slower than the IPM for small problems, and almost twice as fast for the larger one. This is due to the fact that, as the problem size increases, the solution of the KKT system (requiring a number of flops quadratic in the input and state size) gets increasingly cheap with respect to the factorization of the KKT matrix (requiring a cubic number of flops).

However, the accuracy reached in 50 iterations appears to decrease as the problem size increases, and therefore in practice more iterations may be necessary in case of large problems, decreasing the performance advantage. Furthermore, the accuracy of the solution varies considerably with the problem instance.

Table 13.4: Numerical test for the ADMM solver implemented using routines in HPMPC in the solution of the soft-constrained linear MPC problem. Run times are presented in seconds. For each problem size and solver, the number of ADMM iterations is fixed to 50.

M	n_x	n_u	N	n_b	n_s	HPMPC ADMM
2	4	1	10	1	4	$9.32 \cdot 10^{-5}$
4	8	3	10	3	8	$1.55 \cdot 10^{-4}$
6	12	5	30	5	12	$7.49 \cdot 10^{-4}$
11	22	10	10	10	22	$4.23 \cdot 10^{-4}$
15	30	14	10	14	30	$6.18 \cdot 10^{-4}$
30	60	29	30	29	60	$5.74 \cdot 10^{-3}$

13.2.5.2 Soft-constrained linear MPC problems

This section contains the results on the use of the ADMM in the solution of soft-constrained MPC problems. The results are in table 13.4, where the size of the problem is the same as in the IPM case in table 13.2. Also in the soft-constrained case the factorization is performed off-line, and the number of ADMM iterations is fixed to 50.

Generally speaking, in the soft-constrained case the cost per iteration increases only slightly with respect to the box-constrained cases, as a result of the structure exploiting ADMM algorithm. The increase is about 40% for the smaller problems, and 15% for the larger one.

However, the accuracy obtained in 50 iterations is much lower than in the box constrained case. This is significantly different than in the IPM case, where the number of iterations increased only slightly with respect to the box-constrained case. The reference [49] analyzes the reason for the steep increase in the iteration count and suggests proper scaling mitigate the issue. However, this has not been employed in the current tests.

13.3 Conclusion

In this section, IPMs and ADMMs for the solution of linear MPC and MHE problems are reviewed. Efficient implementations are proposed, that make use of the backward Riccati recursion (presented in Section 8.1) for the solution of the

unconstrained sub-problems. Techniques to efficiently process a soft formulation of the constraints is presented, and numerical tests confirm a very small increase in the cost-per-iteration compared to the hard formulation of the constraints.

In particular, IPMs require the factorization and solution of a system of linear equations at each IPM iteration, in the computation of the search direction. The factorization step makes use of level 3 BLAS and LAPACK routines, that, if well optimized, can attain a large fraction of the full FP throughput. Therefore, the performance of IPMs is much more sensitive to the implementation of linear algebra routines than e.g. first order methods (that only use level 2 BLAS routines). The IPM implemented using the linear algebra routines in HPMPC can outperform other state-of-the-art IPMs for optimal control by about an order of magnitude in case of medium to large size MPC problems.

As a conclusion, an IPM implemented using the proposed linear algebra routines is an excellent choice for a solver for linear MPC problems, quickly and reliably finding the solution of linear MPC problems for a wide range of problem sizes and constraint structures.

Chapter 14

Summary and considerations about solution of sub-problems in nonlinear MPC and MHE problems

Part I of the thesis proposed implementation techniques specially tailored to embedded optimization. The focus of the implementation is on getting the best possible performance for small to medium size matrices (i.e. for matrices of size up to a few hundreds). One of the key features is the adoption of a special matrix format (called panel-major), roughly corresponding to the innermost level of packing in the GotoBLAS's approach [44], clearly exposed in [86]. This matrix format provides optimal performance for matrices roughly fitting in the last level of cache, and avoids the need for packing matrices on-line. Furthermore, a specialized linear algebra kernel (based on the `gemm` kernel) is implemented for each linear algebra routine, carefully optimized for a number of computer architectures. The linear algebra routines implemented using these kernels outperform both optimized BLAS libraries and code-generated routines commonly employed in embedded optimization.

Part II of the thesis presented tailored algorithms for the solution of unconstrained MPC and MHE problems. In particular, a backward Riccati recursion is found to be a reliable choice, giving good performance for a wide range of problem sizes, while a forward Schur-complement recursion can handle MPC and MHE problems with further equality constraints on the last stage. Condensing methods for MPC and MHE can be employed as routines in partial condensing, that in turn can be employed as a preparation step to reformulate the MPC and MHE problems in the size giving the best performance for the solver at hand. All these algorithms are implemented using the linear algebra routines developed in Part I of the thesis and therefore all matrices are assumed to be in the panel-major matrix format.

Part III of the thesis quickly reviewed some optimization algorithm that can be implemented using the solvers from Part II to solve the unconstrained sub-problems. In particular, a Riccati-based IPM implemented using the linear algebra routines in HPMPC is found to outperform state-of-the-art solvers for embedded linear MPC problems by more that one order of magnitude for medium to large scale MPC problems. These optimization algorithms still assume that all matrices are in the panel-major matrix format. The conversion between standard row-major or column-major formats into the panel-major format can be performed in a wrapper around the IPM routine. In this way, the conversion overhead is well amortized over a large number of IPM iterations. Numerical tests confirm the effectiveness of the approach.

14.1 Interface with existing solvers for NMPC

The solution of linear MPC and MHE problems is a key step in the solution of NMPC problems using the multiple-shooting and direct collocation discretization methods. In fact, in that case the sub-problems that need to be solved at each Sequential Quadratic Programming (SQP) iteration can be seen as time-variant instances of linear MPC problems [83].

In this section, the strength of the solvers for the MPC and MHE problems in HPMPC are presented for the state estimation and control of a nonlinear system. Namely, the results of closed-loop real-time simulations of rotational start-up for an airborne wind energy system [93] are presented. The system is modeled as a differential-algebraic equation (DAE), with 27 differential states, 1 algebraic state and 4 control inputs.

To solve the NMPC and NMHE formulations, the ACADO Code Generation Tool (CGT) [47, 48] is employed. ACADO implements the real-time iteration

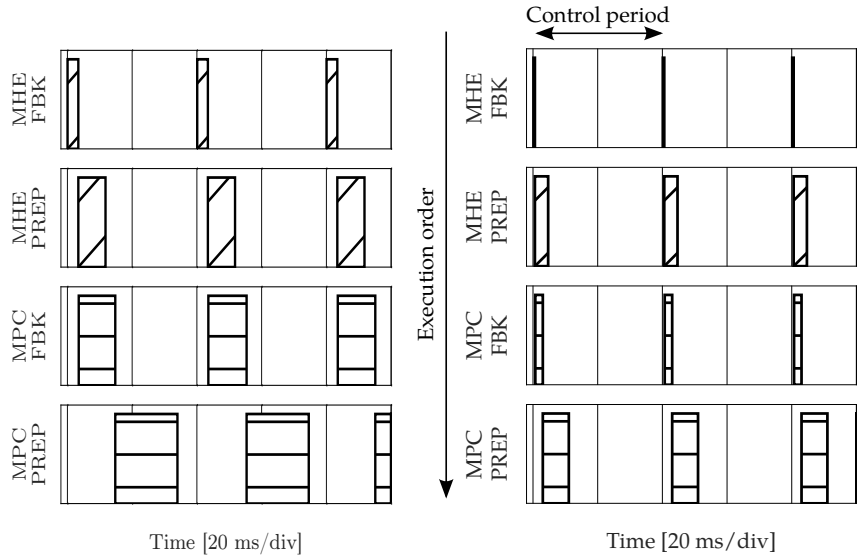
(RTI) scheme [25, 59]. Multiple shooting is employed for the discretization. The QPs underlying the NMPC solver have box constraints, and they are handled with the Riccati-based IPM presented in Section 13.1. The QPs underlying the NMHE solver do not have general or box constraints, but they have further equality constraints on the last stage: they are solved efficiently using the forward Schur-complement recursion presented in Section 8.2. More details about the experimental setup can be found in [88] (where the use of the Riccati-based IPM is investigated for the solution of the MPC sub-problems) and in [40] (where the use of the forward Schur-complement recursion is investigated for the solution of the MHE sub-problems).

The linear MPC sub-problems have $n_x = 27$ states, $n_u = 4$ controls and an horizon length of $N_c = 50$. An augmented model used for the NMHE, one that includes a disturbance model. Therefore the linear MHE sub-problem have $n_x = 33$ states and $n_w = 6$ disturbance inputs. Consistency conditions of the DAE model yield $n_d = 9$ equality constraints on the last stage, while the number of estimation intervals is $N_e = 15$. More details can be found in [87] and references therein.

The simulation results are reported in Figure 14.1. A control interval begins with a feedback step of the RTI scheme for the NMHE (MHE FBK), after which the current state estimate is obtained. Afterwards, the NMPC feedback step is triggered (MPC FBK) for calculation of optimal control inputs. In essence, the execution times of the feedback steps amount to solutions of underlying QPs. After each feedback step corresponding preparation step is executed (MHE PREP and MPC PREP), which includes model integration, sensitivity generation and linearization of the objective and the constraints. In this setting both NMHE and NMPC run on the separate CPU cores.

Figure 14.1a presents the results when the qpOASES [28] solver is used to solve the QPs underlying the NMHE and NMPC formulations. In that case, the feedback step of the NMHE requires about 3.5 ms, while the preparation step requires about 8.5 ms. The feedback step for the NMPC requires about 12 ms, while the preparation step requires slightly less than 20 ms.

Figure 14.1b presents the results when the solvers in HPMPC are employed instead for the solution of the underlying QPs. In that case, the feedback step of the NMHE requires about 0.5 ms (that is about 7 times as fast as in the qpOASES case), while the preparation step requires about 5 ms. The feedback step for the NMPC requires about 3 ms (that is about 4 times as fast as in the qpOASES), while the preparation step requires less than 8 ms. In total, the maximum feedback delay is always less than 3.5 ms, far below the control period of 40 ms.



(a) ACADO-qpOASES. **(b)** ACADO-HPMPC.

Figure 14.1: Feedback and preparation step times for the MHE and MPC using the ACADO-qpOASQS and ACADO-HPMPC solvers in the rotational start-up of an airbone wind energy system.

The much shorter feedback steps of the solvers in HPMPC are due to the fact that the forward Schur-complement method can efficiently solve the exact MHE sub-problem formulations, while the long horizon length of $N_c = 50$ in the MPC sub-problems favors sparse solvers. The much shorter preparation steps are mainly due to the fact that the solvers in HPMPC do not require a condensing step, while that is employed in the qpOASES case. The integration algorithms are identical in the two cases.

Appendix A

Custom gcc Compiler

During the optimization of the `dpotrf` and `dtrtri` kernels on the Haswell micro-architecture, a performance issue has been found in the `gcc` compiler: the performance of these kernels was about 20% lower than expected based on the performance of the other level 3 BLAS kernels.

An analysis of the produced assembly code showed that the `fnmadd` intrinsics (implementing $c \leftarrow c - a \cdot b$, where a, b, c are registers) used in the kernels implementation have been translated into `fmadd` (implementing $c \leftarrow c + a \cdot b$) and `xor` (used to change the sign of either a or b) instructions. This causes a performance degradation due to the introduction of the `xor` instructions in an already resource-constrained loop.

In `gcc`, the intrinsics are implemented as calls to builtin functions, that are functions defined in the compiler itself. Generally, there is a builtin for each intrinsics, but this is not the case for the various `fnmadd`, `fmsub` and `fnmsub` intrinsics. An inspection of the

```
main_gcc_folder\gcc\config\i386\fmaintrin.h
```

header file shows that they are all implemented with a call to the `fmadd` builtin, plus the negation of the proper operands, as (in the case of the 256-bit double-

precision version of the intrinsics, similarly for 128-bit and/or single-precision versions of the intrinsics)

```
extern __inline __m256d
__attribute__((__gnu_inline__, __always_inline__, __artificial__))
_mm256_fmadd_pd (__m256d __A, __m256d __B, __m256d __C)
{
  return (__m256d)__builtin_ia32_vfmaddpd256 ((__v4df)__A, (__v4df)__B,
                                              (__v4df)__C);
}

extern __inline __m256d
__attribute__((__gnu_inline__, __always_inline__, __artificial__))
_mm256_fmsub_pd (__m256d __A, __m256d __B, __m256d __C)
{
  return (__m256d)__builtin_ia32_vfmaddpd256 ((__v4df)__A, (__v4df)__B,
                                              -(__v4df)__C);
}

extern __inline __m256d
__attribute__((__gnu_inline__, __always_inline__, __artificial__))
_mm256_fnmadd_pd (__m256d __A, __m256d __B, __m256d __C)
{
  return (__m256d)__builtin_ia32_vfmaddpd256 (-(__v4df)__A, (__v4df)__B,
                                              (__v4df)__C);
}

extern __inline __m256d
__attribute__((__gnu_inline__, __always_inline__, __artificial__))
_mm256_fnmsub_pd (__m256d __A, __m256d __B, __m256d __C)
{
  return (__m256d)__builtin_ia32_vfmaddpd256 (-(__v4df)__A, (__v4df)__B,
                                              -(__v4df)__C);
}
```

The builtins are defined in the file

```
main_gcc_folder\gcc\config\i386\i386.c
```

where the enumeration `ix86_builtins` contains the codes for all the builtins, that are defined in the `bdesc_multi_arg` array of `builtin_description` structures. The lines

```
  IX86_BUILTIN_VFMSUBPD256,
  IX86_BUILTIN_VFNMADDPD256,
  IX86_BUILTIN_VFNMSUBPD256,
```

have been added to the `ix86_builtins` enumeration, while the lines

```
  { OPTION_MASK_ISA_FMA | OPTION_MASK_ISA_FMA4, CODE_FOR_fma4i_fmsub_v4df,
    "__builtin_ia32_vfmsubpd256", IX86_BUILTIN_VFMSUBPD256,
    UNKNOWN, (int)MULTI_ARG_3_DF2 },
  { OPTION_MASK_ISA_FMA | OPTION_MASK_ISA_FMA4, CODE_FOR_fma4i_fnmadd_v4df,
    "__builtin_ia32_vfnmaddpd256", IX86_BUILTIN_VFNMADDPD256,
    UNKNOWN, (int)MULTI_ARG_3_DF2 },
  { OPTION_MASK_ISA_FMA | OPTION_MASK_ISA_FMA4, CODE_FOR_fma4i_fnmsub_v4df,
    "__builtin_ia32_vfnmsubpd256", IX86_BUILTIN_VFNMSUBPD256,
    UNKNOWN, (int)MULTI_ARG_3_DF2 },
```

have been added to the `bdesc_multi_arg` array. The `builtin_description` structure is defined as

```
struct builtin_description
{
  const HOST_WIDE_INT mask;
  const enum insn_code icode;
  const char *const name;
  const enum ix86_builtins code;
  const enum rtx_code comparison;
  const int flag;
};
```

The file

```
main_gcc_folder\gcc\config\i386\sse.md
```

contains the machine description of the various `SSE`, `AVX` and `FMA` instructions. In particular, it contains a RTL (Register Transfer Language, a low-level intermediate representation) pattern for each instruction that the target machine supports (or that it is worth telling the compiler about). The name of insns from either named `define_insn` or `define_expand` are used to generate a list of insns, that then undergoes various optimization passes. Finally, the insn list's

RTL is matched against the RTL templates in the `define_insn` to produce assembly code.

In the `sse.md` file, there is a `define_insn` for each of the `fmadd`, `fmsub`, `fnmadd` and `fnmsub` instructions, but there is only a single `define_expand`, that corresponds to the single builtin `__builtin_ia32_vfnmaddpd256` defined in the file `i386.c` (that in the struct `builtin_description` has `insn_code` equal to `CODE_FOR_fma4i_fnmadd_v4df`). This means that the compiler can emit all `fmadd`, `fmsub`, `fnmadd` and `fnmsub` instructions if the corresponding patterns are recognized in the optimized list of insns, but that only the `fmadd` builtin can be used to generate the patterns in the list of insns. In practice, the insns generated by the combinations of `fmadd` and `xor` builtins (corresponding to the `fmsub`, `fnmadd` and `fnmsub` intrinsics) undergo the various optimization steps, and their patterns are not recognized as belonging to the `fmsub`, `fnmadd` and `fnmsub` instructions when producing the assembly code.

This is solved by adding a `define_expand` for each of the `fmsub`, `fnmadd` and `fnmsub` builtins defined in `i386.c`. Namely, the lines

```
(define_expand "fma4i_fmsub_<mode>"
  [(set (match_operand:FMAMODE_AVX512 0 "register_operand")
    (fma:FMAMODE_AVX512
      (match_operand:FMAMODE_AVX512   1 "nonimmediate_operand")
      (match_operand:FMAMODE_AVX512   2 "nonimmediate_operand")
      (neg:FMAMODE_AVX512
        (match_operand:FMAMODE_AVX512 3 "nonimmediate_operand"))))])

(define_expand "fma4i_fnmadd_<mode>"
  [(set (match_operand:FMAMODE_AVX512 0 "register_operand")
    (fma:FMAMODE_AVX512
      (neg:FMAMODE_AVX512
        (match_operand:FMAMODE_AVX512 1 "nonimmediate_operand"))
      (match_operand:FMAMODE_AVX512   2 "nonimmediate_operand")
      (match_operand:FMAMODE_AVX512   3 "nonimmediate_operand")))])

(define_expand "fma4i_fnmsub_<mode>"
  [(set (match_operand:FMAMODE_AVX512 0 "register_operand")
    (fma:FMAMODE_AVX512
      (neg:FMAMODE_AVX512
        (match_operand:FMAMODE_AVX512 1 "nonimmediate_operand"))
      (match_operand:FMAMODE_AVX512   2 "nonimmediate_operand")
      (neg:FMAMODE_AVX512
        (match_operand:FMAMODE_AVX512 3 "nonimmediate_operand"))))])
```

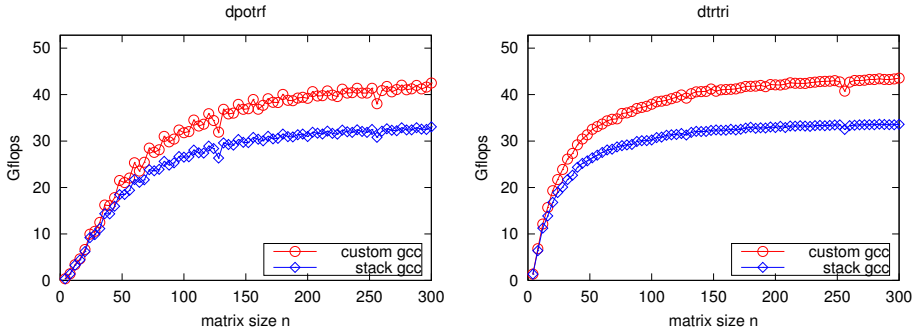
are added to the `sse.md` file.

Finally, the `fmaintrin.h` header can explicitly use the newly defined builtins for the `fmsub`, `fnmadd` and `fnmsub` instructions, as

```
extern __inline __m256d
__attribute__((__gnu_inline__, __always_inline__, __artificial__))
_mm256_fmadd_pd (__m256d __A, __m256d __B, __m256d __C)
{
  return (__m256d)__builtin_ia32_vfmaddpd256 ((__v4df)__A, (__v4df)__B,
                                              (__v4df)__C);
}

extern __inline __m256d
__attribute__((__gnu_inline__, __always_inline__, __artificial__))
_mm256_fmsub_pd (__m256d __A, __m256d __B, __m256d __C)
{
  return (__m256d)__builtin_ia32_vfmsubpd256 ((__v4df)__A, (__v4df)__B,
                                              (__v4df)__C);
}

extern __inline __m256d
__attribute__((__gnu_inline__, __always_inline__, __artificial__))
_mm256_fnmadd_pd (__m256d __A, __m256d __B, __m256d __C)
{
  return (__m256d)__builtin_ia32_vfnmaddpd256 ((__v4df)__A, (__v4df)__B,
                                              (__v4df)__C);
}

extern __inline __m256d
__attribute__((__gnu_inline__, __always_inline__, __artificial__))
_mm256_fnmsub_pd (__m256d __A, __m256d __B, __m256d __C)
{
  return (__m256d)__builtin_ia32_vfnmsubpd256 ((__v4df)__A, (__v4df)__B,
                                              (__v4df)__C);
}
```

In a similar fashion, the builtins for the 128-bit and/or single precision version of the `fmsub`, `fnmadd` and `fnmsub` can be added.

After these modifications, the `gcc` compiler correctly emits the `fnmadd` instructions in the `dpotrf` and `dtrtri` kernels, improving their performance by about 20%. In Figure A.1 there is the comparison of the performance of the `dpotrf`



(a) dpotrf **(b)** dtrtri

Figure A.1: Performance test for routines `dpotrf` and `dtrtri` using stack and curtom `gcc` compilers, on an Intel core i7 4800MQ processor (Haswell micro-architecture, supporting the AVX2 and FMA ISAs).

and `dtrtri` routines when the code is compiled with stack `gcc` compiler (version 5.2.0) and with the custom `gcc` compiler (customization of version 5.2.0).

A modification of the latest `gcc` (version 6.0.0) can be found in [9].

Bibliography

[1] http://www.realworldtech.com/.

[2] http://www.agner.org/optimize/.

[3] ACML. http://developer.amd.com/tools-and-sdks/archive/amd-core-math-library-acml/.

[4] ATLAS. http://math-atlas.sourceforge.net/.

[5] BLAS. http://www.netlib.org/blas/.

[6] BLIS. https://github.com/flame/blis.

[7] CPLEX. http://www.ibm.com/software/integration/optimization/cplex-optimization-studio/.

[8] FORCES_Pro. https://www.embotech.com.

[9] gcc. https://github.com/giaf/gcc.git.

[10] GotoBLAS. https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2.

[11] LAPACK. http://www.netlib.org/lapack/.

[12] MKL. https://software.intel.com/en-us/intel-mkl.

[13] PLASMA. http://icl.cs.utk.edu/plasma/.

[14] BDO Anderson and JB Moore. *Optimal Filtering*. Prentice-Hall, Englewood Cliffs, NJ, 1979.

[15] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' Guide (Third Ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[16] J. Andersson. *A General-Purpose Software Framework for Dynamic Optimization*. PhD thesis, Faculty of Engineering Science, K.U. Leuven, Heverlee, Belgium, 2013.

[17] M. Annergren, A. Hansson, and Wahlberg. B. An ADMM algorithm for solving l_1 regularized MPC. In *IEEE Conference on Decision and Control*, pages 4486–4491. IEEE, 2012.

[18] D. Axehill. Controlling the level of sparsity in MPC. *Systems & Control Letters*, 76:1–7, 2015.

[19] D. Axehill and M. Morari. An alternative use of the Riccati recursion for efficient optimization. *Systems & Control Letters*, 61:37–40, 2012.

[20] A. Bemporad, F. Borelli, and M. Morari. Model predictive control based on linear programming - the explicit solution. *IEEE Transactions on Automatic Control*, 47(12):1974–1985, 2002.

[21] A. Bemporad, M. Morari, V. Dua, and E. N. Pistikopoulos. The Explicit Linear Quadratic Regulator for Constrained Systems. *Automatica*, 38(1):3–20, January 2002.

[22] D.P. Bertsekas. *Dynamic Programming and Optimal Control*, volume 1. Athena Scientific, Belmont, Massachusetts, 1995.

[23] A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and Kurzak J. Mixed precision iterative refinement techniques for the solution of dense linear systems. *The International Journal of High Performance Computing Applications*, 21(4):457–466, 2007.

[24] G. Constantinides. Tutorial paper: Parallel architectures for model predictive control. In *IEEE European Control Conference*, 2009.

[25] M. Diehl. *Real-Time Optimization for Large Scale Nonlinear Process*. PhD thesis, Universität Heidelberg, 2001.

[26] A. Domahidi, A. Zgraggen, M. N. Zeilinger, M. Morari, and C. N. Jones. Efficient interior point methods for multistage problems arising in receding horizon control. In *IEEE Conference on Decision and Control (CDC)*, pages 668 – 674, Maui, HI, USA, December 2012.

[27] I. S. Duff. MA57 - a code for the solution of sparse symmetric definite and indefinite systems. *ACM Trans. Math. Softw.*, 30:118–144, June 2004.

[28] H. J. Ferreau, H. G. Bock, and M. Diehl. An online active set strategy to overcome the limitations of explicit mpc. *International Journal of Robust and Nonlinear Control*, 18(8), 2008.

[29] H. J. Ferreau, C. Kirches, A. Potschka, H. G. Bock, and M. Diehl. qpOASES: A parametric active-set algorithm for quadratic programming. *Mathematical Programming Computation*, 6(4):327–363, 2014.

[30] G. Frison. HPMPC. https://github.com/giaf/hpmpc.git.

[31] G. Frison. Numerical methods for model predictive control. Master's thesis, Department of Informatics and Mathematical Modelling, Technical University of Denmark, Kgs. Lyngby, Denmark, 2012.

[32] G. Frison, H. H. B. Sørensen, B. Dammann, and J. B. Jørgensen. High-performance small-scale solvers for linear model predictive control. In *IEEE European Control Conference*, pages 128–133. IEEE, 2014.

[33] G. Frison and J. B. Jørgensen. Efficient implementation of the riccati recursion for solving linear-quadratic control problems. In *IEEE Multi-conference on Systems and Control*, pages 1117–1122. IEEE, 2013.

[34] G. Frison and J. B. Jørgensen. A fast condensing method for solution of linear-quadratic control problems. In *IEEE Conference on Decision and Control*, pages 7715–7720. IEEE, 2013.

[35] G. Frison and J. B. Jørgensen. Parallel implementation of riccati recursion for solving linear-quadratic control problems. In *18th Nordic Process Control Workshop*, 2013.

[36] G. Frison and J. B. Jørgensen. Efficient Solvers for Soft-Constrained MPC. In *18th Nordic Process Control Workshop*, 2015.

[37] G. Frison and J. B. Jørgensen. MPC related computational capabilities of ARMv7A processors. In *IEEE European Control Conference*. IEEE, 2015.

[38] G. Frison, D. K. M. Kufualor, L. Imsland, and J. B. Jørgensen. Efficient implementation of solvers for linear model predictive control on embedded devices. In *IEEE Multi-conference on Systems and Control*, pages 1954–1959. IEEE, 2014.

[39] G. Frison, L. E. Sokoler, and J. B. Jørgensen. A family of high-performance solvers for linear model predictive control. In *IFAC World Congress*, pages 3074–3079. IFAC, 2014.

[40] G. Frison, M. Vukov, N. K. Poulsen, M. Diehl, and J. B. Jørgensen. High-Performance Small-Scale Solvers for Moving Horizon Estimation. In *IFAC Conference on Nonlinear Model Predictive Control*, pages 80–86. IFAC, 2015.

[41] N. F. Gade-Nielsen, J. B. Jørgensen, and B. Dammann. MPC toolbox with GPU accelerated optimization algorithms. In *10th European Workshop on Advanced Control and Diagnosis*, 2012.

[42] G. H. Golub and C. F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, 3rd edition, 1996.

[43] K. Goto and R. A. van de Geijn. On reducing TLB misses in matrix multiplication. Technical report, Department of Computer Sciences, The University of Texas at Austin, 2002.

[44] K. Goto and R. A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3), 2008.

[45] K. Goto and R. A. van de Geijn. High-performance implementation of the level-3 blas. *ACM Trans. Math. Softw.*, 35(1), 2008.

[46] N. Haverbeke. *Efficient Numerical Methods for Moving Horizon Estimation*. PhD thesis, Katholieke Universiteit Leuven, 2011.

[47] B. Houska, H. J. Ferreau, and M. Diehl. ACADO Toolkit – An Open-Source Framework for Automatic Control and Dynamic Optimization. *Optimal Control Applications and Methods*, 32(3):298–312, 2011.

[48] B. Houska, H. J. Ferreau, and M. Diehl. An Auto-Generated Real-Time Iteration Algorithm for Nonlinear MPC in the Microsecond Range. *Automatica*, 47(10):2279–2285, 2011.

[49] J. Jerez, P. Goulart, S. Richter, G. Constantinides, E. Kerrigan, and M. Morari. Embedded online optimization for model predictive control at megahertz rates. *IEEE Transactions on Automatic Control*, 2013.

[50] J. Jerez, P. Goulart, S. Richter, G. Constantinides, E. C. Kerrigan, and M. Morari. Embedded Predictive Control on an FPGA using the Fast Gradient Method. In *European Control Conference*, pages 3614 – 3620, Zurich, Switzerland, 2013.

[51] J. B. Jørgensen. *Moving Horizon Estimation and Control*. PhD thesis, Department of Chemical Engineering, Technical University of Denmark, Kgs. Lyngby, Denmark, 2005.

[52] J. B. Jørgensen, G. Frison, N. F. Gade-Nielsen, and B. Damman. Numerical methods for solution of the extended linear quadratic control problem. In *IFAC Conference on Nonlinear Model Predictive Control*, pages 187–193. IFAC, 2012.

[53] J. B. Jørgensen, J. K. Huusom, and J. B. Rawlings. Finite horizon MPC for systems in innovation form. In *50th IEEE Conference on Decision and Control and European Control Conference (CDC-ECC)*, pages 1896–1903. IEEE, 2011.

[54] J. B. Jørgensen, J. B. Rawlings, and S. B. Jørgensen. Numerical methods for large-scale moving horizon estimation and control. In *Int. Symposium on Dynamics and Control Process Systems (DYCOPS)*, volume 7, 2004.

[55] T. Kailath, A. H. Sayed, and B. Hassibi. *Linear estimation*. Prentice Hall information and system sciences series. Upper Saddle River, N.J. Prentice Hall, 2000.

[56] R. Kalman. Contributions to the theory of optimal control. *Boletin de la Sociedad Matematica Mexicana*, 1960.

[57] D. K. M. Kufoalor, G. Frison, L. Imsland, T. A. Johansen, and J. B. Jørgensen. Block factorization of step response model predictive control problems. *Jornal of Process Control*, 2015. Submitted.

[58] D. K. M. Kufoalor, S. Richter, L. Imsland, T. A. Johansen, M. Morari, and G. O. Eikrem. Embedded Model Predictive Control on a PLC using a Primal-Dual First-Order Method for a Subsea Separation Process. In *22nd IEEE Mediterranean Conference on Control and Automation*, Palermo, Italy, 2014.

[59] P. Kühl, M. Diehl, T. Kraus, J. P. Schlöder, and H. G. Bock. A real-time algorithm for moving horizon state and parameter estimation. *Computers & Chemical Engineering*, 1(35), 2011.

[60] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Ortí. Analytical models for the BLIS framework. *ACM Transactions on Mathematical Software*, 2015. Pending.

[61] J. M. Maciejowski. *Predictive Control with Constraints*. Prentice Hall, 2002.

[62] J. Mattingley and S. Boyd. CVXGEN: a code generator for embedded convex optimization. *Optimization and Engineering*, 13(1):1–27, March 2012.

[63] D. Q. Mayne, J. B. Rawlings, C. V. Rao, and P. O. M. Scokaert. Constrained model predictive control: Stability and optimality. *Automatica*, 36(6):789–814, 2000.

[64] S. Mehrotra. On the implementation of a primal-dual interior point method. *Journal on Optimization*, 2(4):575–601, 1992.

[65] G. O. Mutambara. *Decentralized estimation and control for multi-sensor systems*. CRC press, 1998.

[66] I. Nielsen, D. Ankelhed, and D. Axehill. Low-rank modifications of Riccati factorizations with applications to model predictive control. In *IEEE Conference on Decision and Control*, pages 3684–3690. IEEE, 2013.

[67] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, New York, 2nd edition, 2006.

[68] B. O'Donoghue, G. Stathopoulos, and S. Boyd. A splitting method for optimal control. *IEEE Transactions on Control Systems Technology*, 21(6):2432–2442, 2013.

[69] S. J. Qin and T. A. Badgwell. A survey of industrial model predictive control technology. *Control Engineering Practice*, 11:733–764, 2003.

[70] C. V. Rao, S. J. Wright, and J. B. Rawlings. Application of interior-point methods to model predictive control. *Journal of optimization theory and applications*, 99:723–757, 1998.

[71] J. B. Rawlings. Tutorial overview of model predictive control. *Control Systems*, 20(3):38–52, 2000.

[72] J. B. Rawlings and D. Q. Mayne. *Model predictive control: theory and design*. Nob Hill Publishing, 2009.

[73] S. Richter, C. N. Jones, and M. Morari. Real-time input-constrained MPC using fast gradient methods. In *IEEE Conference on Decision and Control*, pages 7387 – 7393, 2009.

[74] M. Morari S. Richter and C. N. Jones. Towards Computational Complexity Certification for Constrained MPC Based on Lagrange Relaxation and the Fast Gradient Method. In *IEEE Conference on Decision and Control*, pages 5223 – 5229, 2011.

[75] A. Shahzad, E. C. Kerrigan, and G. A. Constantinides. A fast well-conditioned interior point method for predictive control. In *CDC*, pages 508–513. IEEE, 2010.

[76] L. E. Sokoler, G. Frison, M. S. Andersen, and J. B. Jørgensen. Input-constrained model predictive control via the alternating direction method of multipliers. In *IEEE European Control Conference*, pages 115–120. IEEE, 2014.

[77] L. E. Sokoler, G. Frison, K. Edlund, A. Skajaa, and J. B. Jørgensen. A riccati based homogeneous and self-dual interior-point method for linear economic model predictive control. In *IEEE Multi-conference on Systems and Control*, pages 592–598. IEEE, 2013.

[78] L. E. Sokoler, G. Frison, A. Skajaa, R. Halvgaard, and J. B. Jørgensen. A homogeneous and self-dual interior-point linear programming algorithm for economic model predictive control. *IEEE Transactions on Automatic Control*, 2015.

[79] L. E. Sokoler, A. Skajaa, G. Frison, R. Halvgaard, and J. B. Jørgensen. A warm-started homogeneous and self-dual interior-point method for linear economic model predictive control. In *IEEE Conference on Decision and Control*, pages 3677–3683. IEEE, 2013.

[80] G. Stathopoulos, M. Korda, and C. N. Jones. Solving the infinite-horizon constrained LQR problem using splitting techniques. In *IFAC World Congress*, pages 2285–2290. IFAC, 2014.

[81] G. Stathopoulos, M. Korda, and C. N. Jones. Solving the infinite-horizon constrained LQR problem using accelerated dual proximal methods. ArXiv: 1501.04352, 2015.

[82] G. Stathopoulos, A. Szucs, Y. Pu, and C. N. Jones. Splitting methods in control. In *13th IEEE European Control Conference*, pages 2478–2483. IEEE, 2014.

[83] M. C. Steinbach. *A structured interior point SQP method for nonlinear optimal control problems*. Springer, 1994.

[84] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[85] F. G. Van Zee, T. Smith, F. D. Igual, M. Smelyanskiy, X. Zhang, M. Kistler, V. Austel, J. Gunnels, T. M. Low, B. Marker, L. Killough, and R. A. van de Geijn. The BLIS framework: Experiments in portability. *ACM Transactions on Mathematical Software*, 2015. Accepted.

[86] F. G. Van Zee and R. A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software*, 41(3):14:1–14:33, 2015.

[87] M. Vukov. *Embedded Model Predictive Control and Moving Horizon Estimation for Mechatronics Applications*. PhD thesis, KU Leuven, April 2015.

[88] M. Vukov, S. Gros, G. Horn, G. Frison, K. Geebelen, J. B. Jørgensen, J. Swevers, and M. Diehl. Real-time nonlinear MPC and MHE for a large-scale mechatronic application. *Control Engineering Practice*, 45:64–78, 2015.

[89] Y. Wang and S. Boyd. Fast model predictive control using online optimization. In *IFAC World Congress*, pages 6974 – 6997, Seoul, July 2008.

[90] Y. Wang and S. Boyd. Fast model predictive control using online optimization. *IEEE transactions on Control Systems Technology*, 18(2):267–278, 2010.

[91] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27:2001, 2000.

[92] S. J. Wright. *Primal-dual interior-point methods*. SIAM, 1997.

[93] M. Zanon, S. Gros, and M. Diehl. Rotational Start-up of Tethered Airplanes Based on Nonlinear MPC and MHE. In *Proceedings of the European Control Conference*, 2013.

[94] X. Zhang. OpenBLAS. http://www.openblas.net/.