Simple and efficient GPU accelerated topology optimisation: Codes and applications

Träff, Erik A.; Rydahl, Anton; Karlsson, Sven; Sigmund, Ole; Aage, Niels

Published in:
Computer Methods in Applied Mechanics and Engineering

Link to article, DOI:
10.1016/j.cma.2023.116043

Publication date:
2023

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

Available online at www.sciencedirect.com
ScienceDirect
Comput. Methods Appl. Mech. Engrg. 410 (2023) 116043
Computer methods in applied mechanics and engineering
www.elsevier.com/locate/cma

Simple and efficient GPU accelerated topology optimisation: Codes and applications

Erik A. Träff[a], Anton Rydahl[b], Sven Karlsson[b], Ole Sigmund[a], Niels Aage[a,*]

[a] *Department of Civil and Mechanical Engineering, Technical University of Denmark, Building 404, Koppels Alle, Kgs. Lyngby, 2800, Denmark*
[b] *Department of Applied Mathematics and Computer Science, Technical University of Denmark, Building 324, Richard Petersens Plads, Kgs. Lyngby, 2800, Denmark*

Received 23 October 2022; received in revised form 30 March 2023; accepted 30 March 2023
Available online xxxx

Dataset link: Futhark implementation, OpenMP-GPU implementation, OpenMP-CPU implementation, Futhark Nonlinear implementation

Abstract

This work presents topology optimisation implementations for linear elastic compliance minimisation in three dimensions, accelerated using Graphics Processing Units (GPUs). Three different open-source implementations are presented for linear problems. Two implementations use GPU acceleration, based on either OpenMP 4.5 or the Futhark language to implement the hardware acceleration. Both GPU implementations are based on high level GPU frameworks, and hence, avoid the need for expertise knowledge of e.g. CUDA or OpenCL. The third implementation is a vectorised and multi-threaded CPU code, which is included for reference purposes. It is shown that both GPU accelerated codes are able to solve large-scale topology optimisation problems with 65.5 million elements in approximately 2 h using a single GPU, while the reference implementation takes approximately 3 h and 10 min using 48 CPU cores. Furthermore, it is shown that it is possible to solve nonlinear topology optimisation problems using GPU acceleration, demonstrated by a nonlinear end-compliance optimisation with finite strains and a Neo-Hookean material model discretised by 1 million elements.

Keywords: Topology optimisation; GPU acceleration; Structural optimisation

1. Introduction

Topology optimisation is a rapidly maturing technology which enables automatic design of application tailored structures [1,2]. At the time of writing, this design method has been shown to work for a wide variety of physics. Nevertheless, the computational cost of performing the optimisation is challenging for many practical use-cases. Large-scale topology optimisation problems usually take days to run on expensive compute clusters [3,4], which are only readily available to few large companies and academics. Therefore, the research community seeks to improve the computational efficiency of topology optimisation in order to make the methods easily available to small and medium sized companies. One approach is to use graphics processing units (GPUs) to accelerate the computation.

* Corresponding author.
E-mail address: naage@dtu.dk (N. Aage).

https://doi.org/10.1016/j.cma.2023.116043

Advantages of GPU accelerated computing. Modern GPUs offer a great computing power to price ratio, compared to the more general purpose central processing units (CPUs). Consider the Nvidia 3080Ti GPU, which promises a theoretical limit of 34 TFLOPS on 32-bit floating point numbers. This card was launched in 2021 at a recommended retail price of 1200 USD [5]. A similarly priced CPU, the Intel Xeon W-3335 launched in 2021 with recommended retail price 1300 USD, promises a theoretical limit of 1 TFLOPS on 32-bit floating point numbers (4 GHz $\times$ 16 cores $\times$ 16 SIMD-lanes) [6]. This order of magnitude improvement of computing speeds explains the current interest in using GPUs for computationally intensive tasks outside of graphics processing. It should be noted however, that many graphics cards are optimised for 32-bit floating point performance, as these are commonly used for graphics. The graphics cards with good 64-bit floating point performance are usually high-end cards designed specifically for acceleration of scientific and machine learning workloads. This is a relevant observation as topology optimisation implementations rely on 64-bit, as the high precision is required to avoid truncation errors which arise in the solvers, both for the physics and the optimisation problem [7].

Challenges of general purpose GPU programming. The GPU is able to attain large performance improvements over the CPU by exploiting the large amount of parallelism in graphics processing. The GPU can be thought of as a stream processor, which applies some function, a so-called kernel, to a large set of inputs. It is crucial that implementations for the GPU follow this structure. Most frameworks for general purpose GPU programming, e.g. CUDA or OpenCL, are based on the notion of kernels, in order to ensure that the resulting programs are structured correctly [8,9]. While these frameworks are able to produce very efficient code for the GPU, it is challenging to port CPU programs to GPUs, as the reformulation of the problem to kernels is left to the programmer. One important aspect of the highly parallel GPU architecture is that problems with irregular data access, such as sparse matrix factorisation, are non-trivial to implement efficiently.

An alternate approach to the kernel based model is to use compiler directives to declare loops as kernel executions to be performed by the GPU. This approach is followed by OpenMP and OpenACC which both allow loop bodies to be compiled into kernels as specified by compiler directives [10,11]. Hence, such approaches are interesting when seen from a mechanical engineering perspective.

A practical approach to general purpose GPU programming. An alternat approach to writing explicit kernels is to use a programming language with higher levels of abstraction, which can be compiled to GPU kernels. One such language is Futhark [12–14], which was chosen for this work due to the thoroughness of its language documentation. Similar languages include Dex [15], and Lift [16]. There are several advantages to using Futhark. Writing software at a higher level of abstraction is generally faster. While purely based on perception, it is our experience that using Futhark has significantly reduced the total development time of the presented GPU programs, compared to the corresponding reference program. Furthermore, the Futhark compiler includes an aggressive ahead-of-time optimisation, which results in efficient compute kernels with little effort compared to writing the GPU kernels by hand.

Prior work in topology optimisation using GPUs. The use of general purpose GPU programming for topology optimisation problems goes back more than a decade to Wadbro and Berggren [17] who developed a CUDA implementation to optimise two-dimensional heat conductors. Schmidt and Schulz [18] solved a linear elasticity problem using the conjugate gradient method and were the first to discus the details of writing a GPU kernel for topology optimisation. Further work exploits the structure of Cartesian grids to improve the performance of GPU implementations by introducing multigrid preconditioners with low memory overhead [19–21].

An interesting issue not handled in this work concerns the solution of topology optimisation problems on unstructured meshes. A key challenge for unstructured meshes, is to avoid concurrent writes to a single node, or so-called data-races. One way to avoid data-races is to use graph-colouring, as done in Zegard and Paulino [22],Martínez-Frutos et al. [23]. Recently, it has been considered to build an algebraic multigrid solver on the CPU and transferring it to the GPU, in order to efficiently solve unstructured problems [24].

Summary of this work. In this work we present two GPU accelerated topology optimisation implementations for the linear elastic minimum compliance problem [1], one using the Futhark language, and the other using OpenMP 4.5 to generate GPU kernels. Furthermore, we present a reference implementation for the CPU based on OpenMP for performance comparisons. All presented codes are made publicly available, and it is the authors' hope that the accessibility of such codes will make high resolution topology optimisation a viable option for a larger group of researchers and practitioners than those currently exploiting these possibilities.

Our implementations assume a Cartesian grid and we exploit this structure of the problem to improve the performance. We show that both GPU accelerated implementations for the linear problem are able to solve topology

optimisation problems with more than 50 million hexahedral elements and 100 design iterations in a matter of hours on a single Nvidia A100 GPU [25].

Finally, we demonstrate the extendibility of one of the GPU implementations by solving a nonlinear elasticity problem. To this end we use the Futhark language to solve a nonlinear end-compliance problem [26] using finite strains and a Neo-Hookean material formulation. As in the linear case, a Cartesian grid is used, and exploited for performance in the implementation. The nonlinear equilibrium is solved using a Newton–Krylov approach. The nonlinear problem has a greatly increased computational complexity, due to the need for numerically integrating all element contributions during every matrix-free application of the tangent matrix. We show that it is indeed feasible to solve topology optimisation problems with nonlinear elasticity on the GPU, although it is notably slower than its linear counterpart. This implementation is also made publicly available.

2. Theory and methods

This section is intended to provide the reader with a sufficient overview over the theoretical background of the considered topology optimisation problems.

2.1. Elasticity

Linear Elasticity. Finite element analysis of small strain linear elasticity is well-known and is based on the following strain energy density

$$\phi_{\text{L}} = \frac{\lambda}{2}(E_{kk})^2 + \mu E_{ij}E_{ij}, \tag{1}$$

Where λ and μ denote the Lamé parameters and E_{ij} is the strain tensor. This formulation is thoroughly covered in many textbooks. In this work a tri-linear hexahedral element formulation is used, which can be found in Cook et al. [27]. For the sake of brevity, the formulation is not reproduced in this article. The interested reader is referred to Cook et al. [27].

Nonlinear elasticity. When considering large deformations, Green–Lagrange strains and a Neo-Hookean material model are adopted. The energy expression for the Neo-Hookean formulation is [28]:

$$\phi_{\text{NL}} = \frac{\lambda}{2}(\ln J)^2 + \frac{\mu}{2}(C_{ii} - 3) - \mu \ln J, \quad J = |f_{ij}| \tag{2}$$

Here f_{ij} denotes the deformation gradient tensor, $C_{ij} = f_{ik}f_{kj}$ denotes the Cauchy–Green tensor, and λ and μ the Lamé parameters

$$\lambda = \frac{\nu E}{(1+\nu)(1-2\nu)}, \quad \mu = \frac{E}{2(1+\nu)} \tag{3}$$

In order to stabilise the void elements an interpolation between the nonlinear energy and the linear formulation is used [29]. The interpolation on the element level is given as:

$$\phi_e = E_e\left[\phi_{\text{NL}}(\gamma_e \mathbf{u}_e) + (1-\gamma_e)\phi_{\text{L}}(\mathbf{u}_e)\right] \tag{4}$$

Here ϕ_{L} is the energy corresponding to the linear formulation. Note that the Young's modulus from the Lamé parameters is moved out as a scaling parameter of the entire energy. γ_e is an element-wise interpolation parameter, based on the filtered element density $\tilde{x}_e$, given by

$$\gamma_e = \frac{\tanh(\beta\rho_0) + \tanh(\beta(\tilde{x}_e - \rho_0))}{\tanh(\beta\rho_0) + \tanh(\beta(1-\rho_0))} \tag{5}$$

where the parameters $\beta = 500$ and $\rho_0 = 0.1$ are used for this work. The details of the formulation are given in Appendix.

2.2. Topology optimisation formulation

The chosen topology optimisation problem treated in this work is the classical problem of compliance minimisation with density filtering and a volume constraint. A detailed explanation of the formulations can be found

in Bendsøe and Sigmund [1], Buhl et al. [26] for the linear and nonlinear case respectively. The problems are summarised here for completeness.

Density filter. Ensuring that non-physical phenomena, such as chequer-boarding are avoided, the density filter is applied [30]. The density filter applies a local weighted average to the elements

$$\tilde{x_i} = \frac{\sum_j x_j w_{ij}}{\sum_j w_{ij}}, \quad w_{ij} = \max[r - d_{ij}, 0] \tag{6}$$

Where x is the local element density, r is the filter radius, and d_{ij} is the distance between element centres of element i and j. $\tilde{x}$ denotes the filtered density. Note that w_{ij} only takes non-zero values when $d_{ij} < r$. The element-wise constant densities are preferred as this improves the efficiency of the linear implementation, as element contributions with a constant Young's module are faster to compute.

Stiffness interpolation. In order to avoid intermediate densities in the resulting design, the Solid Isotropic Material Penalisation (SIMP) is used [1]. The interpolation occurs between some solid material stiffness $E_{\max}$, and a low void stiffness $E_{\min}$ chosen so low that it does not have any practical effect on the solution to the elasticity problem. In this work it is chosen such that $E_{\min} = 10^{-6} E_{\max}$. The element-wise interpolation is

$$E_e = E_{\min} + (E_{\max} - E_{\min})\tilde{x_e}^p \tag{7}$$

The penalisation parameter p is chosen to reduce the Young's Module at intermediate values. In this work a value of $p = 3$ is used throughout. For every finite element the local Young's Module is computed by the interpolation. The assembly of local finite element matrices is otherwise unaffected.

Linear optimisation problem. The linear optimisation problem is a compliance minimisation problem for linear elasticity. The problem is analogous to that solved in several Matlab implementations [31,32], with the exception that it is extended to three dimensions. The formal definition of the problem is

$$\begin{aligned}
&\underset{\mathbf{x}\in\mathbb{R}^{n_e}}{\text{minimise}} && \mathbf{u}^\top \mathbf{f} \\
&\text{subject to:} && \\
&\text{filter} && \tilde{\mathbf{x}} = F(\mathbf{x}) \\
&\text{state equation} && \mathbf{r}(\mathbf{u}, \tilde{\mathbf{x}}) = \frac{\partial \Pi}{\partial \mathbf{u}} - \mathbf{f} = 0 \\
&\text{volume} && \frac{1}{\sum_{e=1}^{n_e} v_e} \sum_{e=1}^{n_e} v_e [\tilde{\mathbf{x}}]_e \leq V^* \\
&\text{box} && 0 \leq [\mathbf{x}]_e \leq 1, \quad \forall e \in \{1, 2, \ldots, n_e\}
\end{aligned} \tag{8}$$

Here F is used to denote the elementwise computation of the filter operation from Eq. (6), and Π is used to denote the integral of ϕ in the considered domain and $\mathbf{f}$ is the nodal load vector. For the linear problem we have that $\mathbf{r}(\mathbf{u}, \tilde{\mathbf{x}}) = \mathbf{K}(\tilde{\mathbf{x}})\mathbf{u} - \mathbf{f}$, where $\mathbf{K}(\tilde{\mathbf{x}})$ denotes the assembled stiffness matrix using the given density values to interpolate the Young's module. Note that for the employed rectangular grids, where all elements have identical shape and size, simplifying the volume constraint in practise. The optimisation problem is solved using the Optimality Criteria method [1], similar to the 88-line Matlab code presented in Andreassen et al. [32].

Nonlinear optimisation problem. The nonlinear optimisation problem is in many aspects similar to the linear problem. Both volume constraint, filter technique, and stiffness interpolation are unchanged from the linear problem. The goal of the optimisation process is to minimise the end-compliance, i.e. the compliance of the final deformation [26]. Indeed Eq. (8) is still valid for the nonlinear problem, with the small change of interpretation that $\mathbf{r}(\mathbf{u}, \tilde{\mathbf{x}})$ is no longer a linear system. Instead, a nonlinear problem is solved to find the end-displacement, and a corresponding linear adjoint problem is solved in order to compute the gradients.

In order to compute the sensitivities of the residual, the adjoint method is used to obtain an analytic expression for the gradients. The derivation of the adjoint expression for end-compliance can be found in Buhl et al. [26]. Given a system where equilibrium has been found, the Lagrange multiplier $\boldsymbol{\lambda}$ can be found by

$$\mathbf{K}_t(\mathbf{u}_{\text{equilibrium}})\boldsymbol{\lambda} = \mathbf{f} \tag{9}$$

Where $\mathbf{K}_l(\mathbf{u}_{\text{equilibrium}})$ denotes the tangent matrix in the system of equilibrium. Using the Lagrange multiplier, the sensitivity with respect to an element can be found by

$$\frac{\partial(\mathbf{u}^\top \mathbf{f})}{\partial \rho_e} = \boldsymbol{\lambda}^\top \frac{\partial \mathbf{r}}{\partial \rho_e} \tag{10}$$

The right hand derivative term can be found by taking the partial derivative of Eq. (A.22).

2.3. Solving the linear systems

The multigrid preconditioned conjugate gradient method is used to solve the linear systems, arising from both linear elasticity, as well as the linearisations of the nonlinear elastic problem. The method is widely used in large-scale topology optimisation problems [3,33], as its error smoothing properties allow for an efficient solution of a design iteration by reusing the previous solution as an initial guess. In this work a V-cycle multigrid is employed as the preconditioner for the conjugate gradient method.

Conjugate gradient. The preconditioned conjugate gradient method is used to solve the linear problems. The method is well established, and details can be found in e.g. Saad [34]. For the linear elastic problem the mixed precision scheme presented in Liu et al. [7] is used. 32-bit floating point numbers are used to store the auxiliary vectors, while the matrix operations and multigrid hierarchy are computed using 64-bit floating point numbers.

The conjugate gradient method requires that the system matrix solved is symmetric positive definite (SPD). While this is always the case for linear elasticity, it is possible to generate non-SPD matrices in the nonlinear case when instabilities occur in the solution. In practice this does not happen unless the magnitude of the applied load is too high compared to the load-carrying ability of the structure. To remedy this potential issue a load continuation scheme, as discussed in Section 3, is introduced during the early design iterations.

Multigrid preconditioner. A V-cycle multigrid, shown in algorithm 1, is used as the preconditioner for the conjugate gradient method. The prolongation matrices P_l are used to project the vectors and matrices between grids. These prolongation matrices are constructed by using the element shape-functions to point-wise evaluate the coarse grid value at the nodes of the finer mesh. In practise the operators P_l and K_l are never assembled in matrix form, but are implemented through functions, with the exception of K_l for the coarsest grids.

For the OpenMP implementations, the used coarse space matrices are not the Galerkin projections shown in algorithm 1 lines 1–3. Instead, the linear elastic stiffness matrix is assembled on the coarse grid, with spatially varying densities within each tri-linear hexahedral element, similar to the approach used in Nguyen et al. [35]. In practice, this means that the coarse grid matrices are assembled by considering the larger coarse elements with a multiresolution density described by the finest level density distribution. The element formulation itself is unchanged from Nguyen et al. [35]. This approach gives rise to a slightly worse preconditioner, but a much simpler implementation. It was therefore chosen for the OpenMP codes, to reduce their inherent complexity.

Algorithm 1: Multigrid V-cycle

Data: Initial residual b_0, Stiffness matrix K_0
Result: smoothed displacement u_0

1 **for** l=1,...,L **do**
2 $\quad K_l \leftarrow P_l^\top K_{l-1} P_l$; `// construct coarse space matrices`
3 **end**
4 **for** l=0,1,...,L-1 **do**
5 $\quad u_l \leftarrow \text{SSOR}(0, b_l, K_l)$; `// smooth`
6 $\quad r_l \leftarrow b_l - K_l u_l$; `// compute residual`
7 $\quad b_{l+1} \leftarrow P_{l+1\top} r_l$; `// restrict residual`
8 **end**
9 Solve $K_L u_L = b_L$ as tight as possible ; `// solve coarse space`
10 **for** l=L-1,L-2,...,0 **do**
11 $\quad u_l \leftarrow u_l + P_l u_{l+1}$; `// prolong solution estimate`
12 $\quad u_l \leftarrow \text{SSOR}(u_l, b_l, K_l)$; `// smooth`
13 **end**

Smoothing. The used multigrid preconditioner uses nodal symmetric successive over-relaxation (SSOR) iterations to smooth the residual at all levels. This smoother is similar to the one used in Wu et al. [36], although it is extended to retain symmetry properties.

Algorithm 2: Nodal symmetric successive over-relaxation

Data: displacement vector $\mathbf{u}$, force vector $\mathbf{f}$, stiffness matrix $\mathbf{K}$, damping parameter $\omega(= 0.6)$
Result: smoothed displacement $\hat{\mathbf{u}}$

$\mathbf{s} \leftarrow \mathbf{Ku}$; // apply stiffness matrix
$\hat{\mathbf{u}} \leftarrow \mathbf{u}$;
forall *nodes n in mesh* **do** // Update independently
  $M \leftarrow \mathbf{K}^n$; // nodal 3x3 matrix
  $r \leftarrow \mathbf{s}^n - Mu^n$;
  $\hat{\mathbf{u}}_1^n \leftarrow \frac{1}{M_{11}}\left(\mathbf{f}_1^n - r_1 - M_{12}\hat{\mathbf{u}}_2 - M_{13}\hat{\mathbf{u}}_3\right)$;
  $\hat{\mathbf{u}}_2^n \leftarrow \frac{1}{M_{22}}\left(\mathbf{f}_2^n - r_2 - M_{21}\hat{\mathbf{u}}_1 - M_{23}\hat{\mathbf{u}}_3\right)$;
  $\hat{\mathbf{u}}_3^n \leftarrow \frac{1}{M_{33}}\left(\mathbf{f}_3^n - r_3 - M_{31}\hat{\mathbf{u}}_1 - M_{32}\hat{\mathbf{u}}_2\right)$;
end
$\mathbf{u} \leftarrow \omega\hat{\mathbf{u}} + (1-\omega)\mathbf{u}$; // Damped update
$\hat{\mathbf{u}} \leftarrow \mathbf{u}$;
$\mathbf{s} \leftarrow \mathbf{Ku}$;
forall *nodes n* **do** // Update independently
  $M \leftarrow \mathbf{K}^n$;
  $r \leftarrow \mathbf{s}^n - Mu^n$;
  $\hat{\mathbf{u}}_3^n \leftarrow \frac{1}{M_{33}}\left(\mathbf{f}_3^n - r_3 - M_{31}\hat{\mathbf{u}}_1 - M_{32}\hat{\mathbf{u}}_2\right)$;
  $\hat{\mathbf{u}}_2^n \leftarrow \frac{1}{M_{22}}\left(\mathbf{f}_2^n - r_2 - M_{21}\hat{\mathbf{u}}_1 - M_{23}\hat{\mathbf{u}}_3\right)$;
  $\hat{\mathbf{u}}_1^n \leftarrow \frac{1}{M_{11}}\left(\mathbf{f}_1^n - r_1 - M_{12}\hat{\mathbf{u}}_2 - M_{13}\hat{\mathbf{u}}_3\right)$;
end
$\hat{\mathbf{u}} \leftarrow \omega\hat{\mathbf{u}} + (1-\omega)\mathbf{u}$;

Algorithm 2 shows the nodal SSOR smoothing as implemented for the multigrid. Here the superscript $(\cdot)^n$ is used to describe accessing the local vector of size 3, or 3×3 matrix, associated with node n. The nodal SSOR is applied successively for several smoothing sweeps, in this work two sweeps are always used unless otherwise stated. The damping parameter ω is chosen as a constant 0.6 in this work, as this is found to perform consistently well through trials. The OpenMP implementations use the simpler Jacobi iteration to smooth the residual on intermediate levels.

Coarse space correction. The coarse space correction is solved approximately using the Jacobi preconditioned conjugate gradient method. Ideally a direct solver should be employed, as the coarse problem is typically so small that a direct solver yields the best results. Unfortunately, a sparse matrix direct solver is currently not available in the Futhark ecosystem, and therefore an alternative approach was necessary.

Similar to the PETSc based topology optimisation framework from Aage et al. [3], the Futhark implementation uses yet another Krylov method to solve the coarse space correction. Specifically, the Futhark implementation uses the Jacobi preconditioned conjugate gradient due to memory efficiency, and due to ease of implementation. The conjugate gradient method is run as a coarse space correction for 800 iterations (4000 iterations in the nonlinear case), or until a relative tolerance of 10^{-10} is achieved. The OpenMP implementations, on the other hand, use a direct solution strategy for the coarsest level, using the Cholmod package to implement the factorisation and back substitution [37].

3. Implementation

This section is intended to provide the reader with a sufficient overview over the three linear elastic implementations in question, their differences, complexities, and limitations. However, as none of the presented code frameworks are shorter than 100 lines, cf. the multitude of Matlab codes, it is not possible to go through the frameworks line by line. Instead the most important and crucial parts are highlighted and explained in detail. The three implementations follow very similar structures and, unless otherwise noted, details given in this section are valid

for all implementations. Two slower reference implementations written in Matlab are also present in the OpenMP repositories, which can be used to compare with the provided implementations. These Matlab implementations are not discussed further in this work, as they are significantly less efficient than the provided OpenMP-CPU implementation. The three implementations are denoted as follows.

Futhark The first GPU accelerated implementation is developed in the Futhark language [12]. The Futhark implementation for the linear problem is available at doi:10.5281/zenodo.7791871.

OpenMP-GPU The second implementation is written in the C language with OpenMP 4.5, and uses GPU acceleration through OpenMP. The OpenMP-GPU implementation is available at doi:10.5281/zenodo.7791868.

OpenMP-CPU Finally, a version of the OpenMP implementation which only uses OpenMP to parallelise the work on the CPU is presented, mostly as a reference code. This implementation is available at doi:10.5281/zenodo.7791870.

Remark, that the exact details of the solver setup differs between the Futhark implementation and the two OpenMP based implementations. This is because there is no direct factorisation implementation available for sparse matrices in Futhark, and the coarse space correction is therefore only computed approximately using the conjugate gradient algorithm [38]. In principle, it would be possible to split the futhark kernels in a way that allowed a direct solver to be employed on the coarse level, but this was not chosen, as it would introduce a lot of complexity to the implementation, and hence, it was deemed outside the scope of this work which attempts striking a balance between simplicity and performance. In contrast, the OpenMP implementation has easy access to any direct solver which has a C interface. It was chosen to use the best possible solver setup for each implementation, in order to avoid restricting the performance artificially. As the multigrid method generally performs better with a direct solution on the coarse space, it was chosen for the OpenMP implementations. The direct solver comes at the cost of transferring data between the CPU and GPU every time the preconditioner is applied. We nevertheless found that the total number of used iterations was reduced, resulting in an overall reduction of compute time. The second and final difference between the Futhark and OpenMP implementations is that the Futhark implementation uses nodal successive over-relaxation for smoothing the residual between multigrid levels, while the OpenMP implementations use Jacobi smoothing. SSOR was only implemented for Futhark, as the high-level language eased writing an additional smoother. While the SSOR implementation was found to perform better, it was not ported to the OpenMP based codes, due to a consideration of the improvement of run-time compared to the time required for implementation.

Matrix-free operator. In order to update all nodes in the finite element mesh independently, a nodal traversal approach is used. An informal way to describe the update $f = Ku$ is by

$$f_i^n = \sum_{e \in \mathcal{E}_n} E_e \sum_{j=1}^{24} K_{ij}^{e,n} u_j^e, \quad i \in \{1, 2, 3\} \tag{11}$$

where f_i^n denotes a component of the resulting force vector at node n, $\mathcal{E}_n$ denotes the set of neighbouring elements of the node n, E_e denotes the Young's modulus associated with element e, $K_{ij}^{e,n}$ denotes a 3×24 slice of the preintegrated stiffness matrix, where the three rows correspond to the node n, and u_j^e denote the local deformations associated with the nodes of element e.

While this approach requires additional work, as u_j^e and E_e are computed once for every neighbouring node, we can parallelise the updates across nodes, without the usual problem of potential data-races if the nodal values are updated element-by-element. In order to apply the matrix-free operator for a coarse space, such as the next-to finest mesh, the nodal values are prolonged to the fine mesh, and the fine mesh matrix-free operator is applied. The resulting values are then restricted to the original mesh. This is in contrast to most standard FEM implementations. The reader is referred to Schmidt and Schulz [18],Wu et al. [19] which treat the subject of nodal updates more thoroughly.

Assembly of coarse operators. For the coarse levels in the multigrid preconditioner, the Galerkin projection of the matrices is constructed and stored in memory. The computation of the matrices is shown in algorithm 1 and an example for level 2 is given in Eq. (12).

$$K_2 = P_2^\top P_1^\top K_0 P_1 P_2 \tag{12}$$

The matrices cannot be computed using sparse matrix–matrix products, as the matrices for the finest levels cannot be stored in memory. Instead, the coarse matrices are computed by repeatedly applying vectors from the standard basis of the coarse space. As an example, the first column of the assembled matrix at level 2 can be extracted by applying matrices of the left hand side of Eq. (12) to $\mathbf{e}_1$, i.e. the basis vector with value 1 in the first entry and 0 in all other entries. By repeatedly prolonging the values using matrix-free representations of P_l, then applying the matrix-free fine level operator K_0, and finally restricting the resulting values back to the coarse mesh. The locality of the prolongation and restriction can be used to only compute fine values in a small neighbourhood around the node which is being expanded, which is needed for making this approach feasible. For the alternate coarse space matrix formulation used in the OpenMP implementations, a usual finite element assembly is used for the coarse space matrices as described in Section 2.3.

3.1. Comparison of languages

In order to compare the complexity of the different implementations, we show the implementation of the density filter in both Futhark and C with OpenMP in Listing 1 and 2. The purpose of this comparison is to show the differences in programming styles, and ease of use. The interested reader may compare these implementations with the simpler `convolutionTexture` examples [39] and the optimised `convolutionSeparable` [40] example provided by NVIDIA, which both implement a separable convolution in two dimensions. As the density filter on a structured grid is also a separable convolution, albeit in three dimensions and with a slightly more complex filter kernel, these implementations give a good indication of how an equivalent CUDA implementation might look.

We note that there is a trade-off for performance in the choice of high-level languages. The simplicity comes at a loss of control of the exact layout of memory, thread groupings, and more. Therefore, a carefully written and tuned lower-level implementation will most probably outperform the presented implementations. Furthermore, high-level languages do not absolve the user from knowing some architectural details to achieve the best performance, such as e.g. having to set an appropriate stencil size in the OpenMP implementations to match available parallelism.

The Futhark implementation in listing 1 is based on a functional programming language, which uses higher order functions. The concept of this implementation is to compute the two sums from Eq. (6) independently, before computing the filtered density value. Initially a series of helping methods and types are defined on lines 1–32. Specially noteworthy is the `sumOverNeighbourhood`, which is a so-called higher-order function that evaluates an input function for all neighbouring elements, and sums the results. This is a core operation in the density filter, which is used to compute both the sum of weights (line 40), and the sum of the scaled densities (line 41). The implementation of `sumOverNeighbourhood` makes use of pipes (written `|>`) which work much like Unix pipes, passing the output of the left expression as input to the function to the right. Another noteworthy detail is the partial application of functions, which is used to create new function, seen in lines 37, 40, and 41. Here the first input values of a function are passed, to create a new function that takes the remaining input.

```
1  type index = {x: i64, y: i64, z: i64}
2
3  -- Check if index is valid.
4  let isInsideDomain nelx nely nelz (idx :index) :bool =
5   idx.x >= 0 && idx.y >= 0 && idx.z >= 0 &&
6   idx.x < nelx && idx.y < nely && idx.z < nelz
7
8  -- Compute weight given radius, element, and neighbour.
9  let getFilterWeight rmin (ownIdx :index) (neighIdx :index) =
10  f32.max 0 (rmin - f32.sqrt( f32.i64 (
11    (ownIdx.x-neighIdx.x)*(ownIdx.x-neighIdx.x) +
12    (ownIdx.y-neighIdx.y)*(ownIdx.y-neighIdx.y) +
13    (ownIdx.z-neighIdx.z)*(ownIdx.z-neighIdx.z))))
14
15 -- Sums the output of computeValue in a local neighbourhood with radius boxRadius of
       element [i,j,k].
16 let sumOverNeighbourhood boxRadius nelx nely nelz i j k (computeValue :index -> f32) =
17  let boxSize = 2*boxRadius+1
18  in tabulate_3d boxSize boxSize boxSize (\ii jj kk ->
19   let neighIdx :index = {x=i+ii-boxRadius,y=j+jj-boxRadius,z=k+kk-boxRadius}
```

```
20    in
21     if isInsideDomain nelx nely nelz neighIdx
22      then computeValue neighIdx
23      else 0)
24  |> map (map f32.sum)
25  |> map f32.sum
26  |> f32.sum
27
28 -- given origin and neighbour indices, compute the weighted density contribution.
29 let getScaledDensity (x :[][][]f32) rmin ownIdx neighIdx =
30  let wgt = getFilterWeight rmin ownIdx neighIdx
31  let dens = #[unsafe] x[neighIdx.x,neighIdx.y,neighIdx.z]
32  in wgt*dens
33
34 -- For all elements, compute the filtered density.
35 entry forwardDensityFilter [nelx][nely][nelz] (rmin :f32) (x :[nelx][nely][nelz]f32) :[nelx
       ][nely][nelz]f32 =
36  let boxRadius = i64.max 0 (i64.f32 (f32.ceil (rmin-1)))
37  let sumOverThisNeighbourhood = sumOverNeighbourhood boxRadius nelx nely nelz
38  in tabulate_3d nelx nely nelz (\i j k->
39   let ownIdx :index = {x=i,y=j,z=k}
40   let weightSum = sumOverThisNeighbourhood i j k (getFilterWeight rmin ownIdx)
41   let scaledDensity = sumOverThisNeighbourhood i j k (getScaledDensity x rmin ownIdx)
42   in scaledDensity / weightSum)
```

Listing 1: Density filter implementation in Futhark

The C implementation using OpenMP in listing 2 is based on a slightly different algorithm for computing the density filter. Instead of adding 0 values for the out-of-bounds neighbours, as done in the Futhark implementation, their contributions are not computed, by adjusting the size of the inner loop. The C implementations have domain padding, changing the loop limits (line 10–12), and complicating the computation of indices slightly (lines 13,30). A `gridContext` data-structure is used to store grid dimensions, including information on the padding. The domain padding is added to allow an efficient implementation of the stiffness matrix operator, which is one of the most time-consuming methods. While the density filter would be faster if a style like the Futhark implementation was used, this was not prioritised, as the overall program time spent in the density filter is low.

```
1  void applyDensityFilter(const struct gridContext gc, const DTYPE rmin, const DTYPE *rho,
       DTYPE *out) {
2   const uint32_t nelx = gc.nelx;
3   const uint32_t nely = gc.nely;
4   const uint32_t nelz = gc.nelz;
5
6   const uint32_t elWrapy = gc.wrapy - 1;
7   const uint32_t elWrapz = gc.wrapz - 1;
8
9   #pragma omp target teams distribute parallel for collapse(3) default(none) firstprivate(
       nelx,nely,nelz,rmin,elWrapy,elWrapz) shared(out,rho)
10  for (unsigned int i1 = 1; i1 < nelx + 1; i1++)
11   for (unsigned int k1 = 1; k1 < nelz + 1; k1++)
12    for (unsigned int j1 = 1; j1 < nely + 1; j1++) {
13     const uint32_t e1 = i1 * elWrapy * elWrapz + k1 * elWrapy + j1;
14     double oute1 = 0.0;
15     double unityScale = 0.0;
16
17     // loop over neighbourhood
18     const uint32_t i2max = MIN(i1 + (ceil(rmin) + 1), nelx + 1);
19     const uint32_t i2min = MAX(i1 - (ceil(rmin) - 1), 1);
20
21     for (uint32_t i2 = i2min; i2 < i2max; i2++) {
22      const uint32_t k2max = MIN(k1 + (ceil(rmin) + 1), nelz + 1);
23      const uint32_t k2min = MAX(k1 - (ceil(rmin) - 1), 1);
```

```
24
25      for (uint32_t k2 = k2min; k2 < k2max; k2++) {
26       const uint32_t j2max = MIN(j1 + (ceil(rmin) + 1), nely + 1);
27       const uint32_t j2min = MAX(j1 - (ceil(rmin) - 1), 1);
28
29       for (uint32_t j2 = j2min; j2 < j2max; j2++) {
30        const uint32_t e2 = i2 * elWrapy * elWrapz + k2 * elWrapy + j2;
31        const double filterWeight = MAX(0.0, rmin - sqrt((i1 - i2) * (i1 - i2) + (j1 - j2) *
      (j1 - j2) + (k1 - k2) * (k1 - k2)));
32
33        oute1 += filterWeight * rho[e2];
34        unityScale += filterWeight;
35       }
36      }
37     }
38     out[e1] = oute1 / unityScale;
39    }
40 }
```

Listing 2: Density filter implementation in C with OpenMP

The OpenMP pragma which parallelises the density filter is shown in line 9. To get the corresponding multi-threaded CPU implementation, the keywords `target teams distribute` should be removed. This is a standard pragma that compiles the loop-body to a GPU kernel, with a `collapse` keyword to indicate that it is the iteration space of all three loops which must be distributed. The `firstprivate` clause is used to indicate that thread-local copies of the variable are created, initialised with the current value of the variable. The `shared` clause indicates that these variables are accessible by all threads, and that the programmer is responsible to avoid data-races.

While the Futhark implementation is about the same length as the C implementation, it has several advantages. By moving the summation out as a higher order function, the filtering method itself is written very concisely, and other similar methods, such as the filtering of the gradients using the chain rule, can use the same set of helping functions. While helping functions are also possible to implement in C, the implementation of local neighbourhood summation would quickly become complex and prone to mistakes, due to the use of function pointers. Furthermore, the strict type system in Futhark allows many types to be inferred by the compiler, and ensures that there is no unexpected behaviour due to implicit type conversions. As a partial summary, the authors infer that Futhark is in many ways simpler to work with than C/OpenMP, despite the additional need to learn the Futhark language in the first place.

When comparing these implementations to the CUDA convolution examples [39,40], it can be seen that the CUDA examples are significantly longer, even though they solve a simpler problem. When trimming the CUDA programs for headers and such, it can be seen that the simple and optimised implementations are 130 and 175 lines respectively, compared to the around 40 lines for both Futhark and OpenMP in listings 1 and 2. The optimised kernel splits the computation into blocks, adds a halo, unrolls loops, and uses more of such transformations to improve the performance. While this is necessary to achieve the best performance on a GPU, it also decreases readability and ease of understanding of the implementation.

Tuning the OpenMP-GPU Implementation. In general, GPUs excel at doing simple, identical operations on many pieces of data simultaneously. This is known as Single Program Multiple Data (SPMD). In principle, adding OpenMP directives for GPU acceleration to an existing C-code is a simple process. However, it is required to refactor some loops to increase the level of parallelism and to reduce the number of registers used in the loop bodies to achieve efficient kernels.

As copying data between the CPU and the GPU introduces overhead, it is important to think about where data resides and when it is transferred. Determining which parts of the program should run on the host and the accelerator, therefore, has to be decided by inspecting profiling results for various problem sizes. Due to the overhead of transferring data, it may be beneficial to run some inefficient kernels on the GPU if that can bring down the number of transfers.

As an example, the density filter kernel in listing 2 is rather complex and does not achieve high streaming-multiprocessor throughput due to non-uniform access patterns. However, by running it on the GPU we can limit the number of data transfers. Further, it is important to verify that the `map` clause has the intended behaviour. When

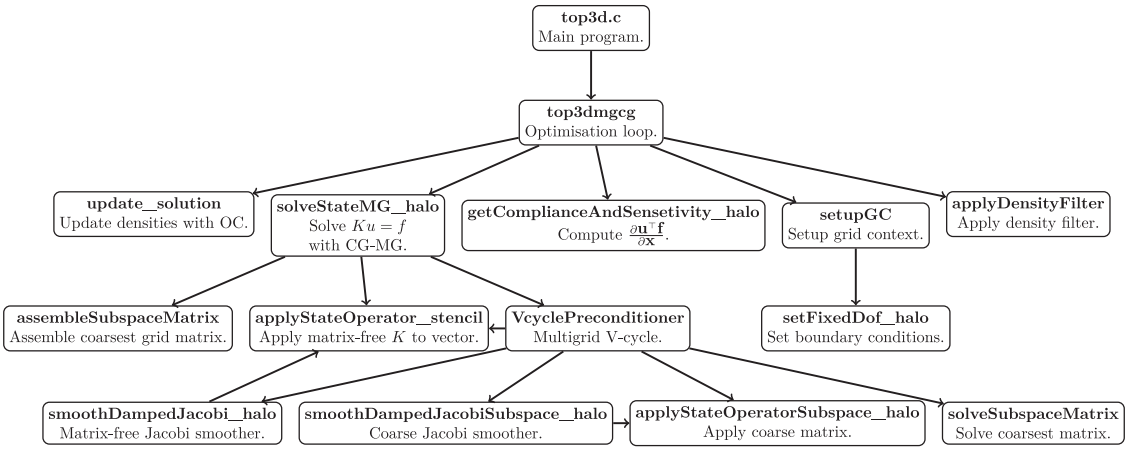


Fig. 1. Program structure of the OpenMP-GPU implementation. Each node represents a significant function.

doing a reduction on the GPU it may be needed to use the `always,tofrom` map modifier for the reduction variable. Otherwise, the reduction variable may be uninitialised on the GPU and the result may not be copied back to the CPU. That depends on whether or not the reduction variable has already been mapped. It is also important to ensure that the environment variable `OMP_TARGET_OFFLOAD` is set to `MANDATORY`. Otherwise, bugs can be introduced if a kernel may be executed on the host instead of the accelerator while the result is transferred back from the accelerator.

3.2. OpenMP implementation structure

The two OpenMP based implementations are structured similarly and are therefore both covered in this section. The program execution begins in `main.c`, which reads parameters from the command line, and calls the optimisation routine. The main parts of the implementation are described in this section. A sketch of the most important functions, and their calling sequence is presented in Fig. 1, to help navigate the implementation.

System definitions. The header `definitions.h` is used to set several important system-wide parameters, which are used by the entire program. Most importantly, the stencil size and floating point types are defined here. The stencil size indicates the amount of nodes in the finite element mesh are updated simultaneously. For the CPU version best performance is achieved by selecting the number of double precision floating points which fit in the vector instructions of the machine, which are 4 for AVX2, or 8 for AVX512. The default value is set to 8, which ensures acceptable performance on all current CPUs. For the GPU version best performance is usually achieved with 64 or 128, depending on the used GPU. Again, the default value is set to the higher value of 128 to ensure acceptable performance everywhere.

`definitions.h` also defines the `gridContext` struct, which is used to store all necessary grid data, such as local element matrices, material parameters, and boundary conditions. Additional methods to use the struct are defined in `grid_utilities.h`, such as the initialisation utility **setupGC**. This is also where the boundary conditions are defined in the function **setFixedDof_halo**, and where they can be modified.

Optimisation. `stencil_optimization.c` is the central file for the optimisation process, where the main optimisation loop is defined in the **top3dmgcg** method. Optimisation specific methods, such as filtering (**applyDensityFilter**), computation of sensitivities (**getComplianceAndSensitivity_halo**), and the optimality criteria update (**update_solution**) are also implemented here.

Multigrid solver. The multigrid solver is implemented in the `multigrid_solver.c` file, which includes the conjugate gradient method for the fine level (**solveStateMG_halo**), and the multigrid preconditioner itself (**VcyclePreconditioner**). Grid operations, such as the matrix-free stiffness matrix product for the fine level (**applyStateOperator_stencil**), matrix-free stiffness matrix product for the coarse level (**applyStateOperatorSubspace_halo**), and prolongations between grids are defined in `stencil_methods.c`, using utility methods from `stencil_utility.h`. The Cholmod [37] direct solver for the coarse grid

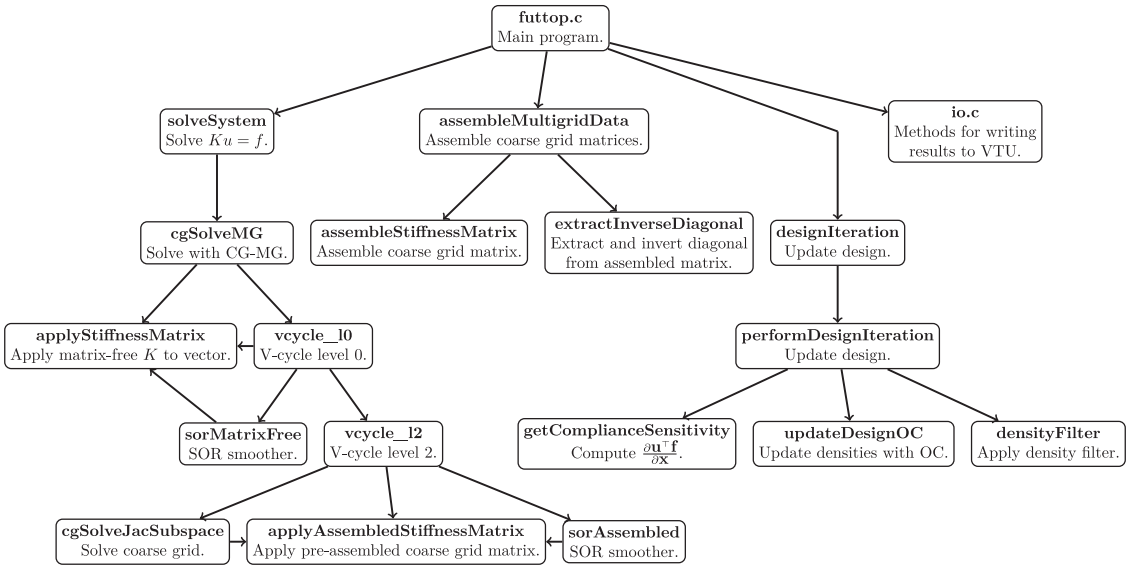


Fig. 2. Program structure of the Futhark implementation. Each node represents a significant function.

(**solveSubspaceMatrix**) is called in `coarse_solver.c`, and the assembly of the coarse grid matrix is defined in `coarse_assembly.c`.

Utilities. Additional utility methods are also included in separate files. `local_matrix.c` defines the integration of the local matrices, which is performed once before starting the program. `write_vtk.c` defines methods to write the result to a vtk file. A small benchmark suite is also present in `benchmark.cpp`, using the Google Benchmark library.

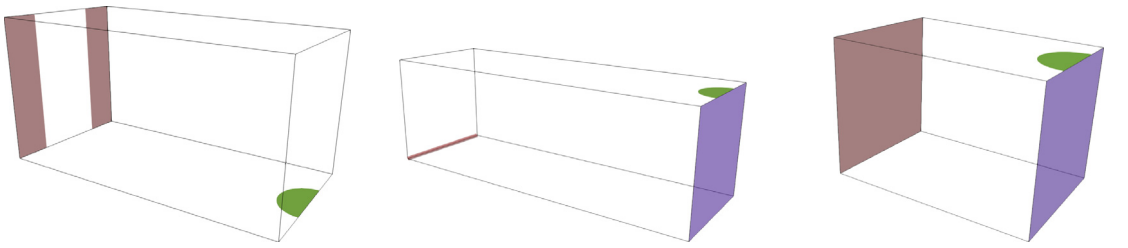
3.3. Futhark implementation structure

The Futhark implementation is structured differently from the OpenMP implementation, due to the functional nature of the Futhark language. This structure is sketched in Fig. 2, to help navigate the implementation. The execution begins in the `futtop.c` file, which contains the main part of the program. The GPU kernels which are called from the C code are defined in the `libmultigrid.fut` source file, which imports all necessary Futhark sources.

Optimisation. The main design loop is implemented in `futtop.c`, which calls four different GPU kernels, all defined in `libmultigrid.fut`. These kernels apply the density filter, assemble the coarse space matrices (**assembleMultigridData**), solve the linear system (**solveSystem**), and update the design variables (**designIteration**). The choice was made to split the functionality into multiple kernels, as this improves the compilation time of the Futhark compiler. Updating the design variables with the optimality criteria method is implemented in the **updateDesignOC** method, defined in `optimization.fut`. The forward and backward filtering are implemented in `densityFilter.fut`.

Multigrid solver. The preconditioned conjugate gradient solver **cgSolveMG** is defined in `solver.fut`, while the multigrid preconditioner **vcycle_l0** is defined in `multigrid.fut`. Similarly, `projection.fut` implements the projection between multigrid levels, `sor.fut` implements the smoothing operations **sorMatrixFree** and **sorAssembled**, `assembly.fut` implements the assembly of coarse matrices, and so on. Of special interest is **applyStiffnessMatrix** defined in `applyStiffnessMatrix.fut`, which implements the matrix-free stiffness matrix product, and `boundaryConditions.fut` which define the boundary conditions for the problem, which are then called by **applyStiffnessMatrix**.

Utilities. The program makes use of several utility methods, which helps keep the code concise. These are defined in `utilities.fut` for general array utilities, and `indexUtilities.fut` for indices and element numbering. Precomputed matrices are also stored in the separate source files `keConstants.fut` and `assemblyWeights.fut`. Methods to write VTK files are implemented in `io.c`.



(a) Cantilever. (b) MBB beam. (c) Double clamped beam.

Fig. 3. Domains for the presented examples in this article. Purple is used to denote a surface with a symmetry condition normal to the surface. Green is used to indicate the position of the traction loads. Brown is used the indicate the position of the Dirichlet boundary conditions. For (a) and (c) red indicates a clamped surface. For (b) brown indicates that the displacement is 0 in the vertical direction, furthermore one point on the brown surface in (b) is prescribed a zero displacement in the y direction. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

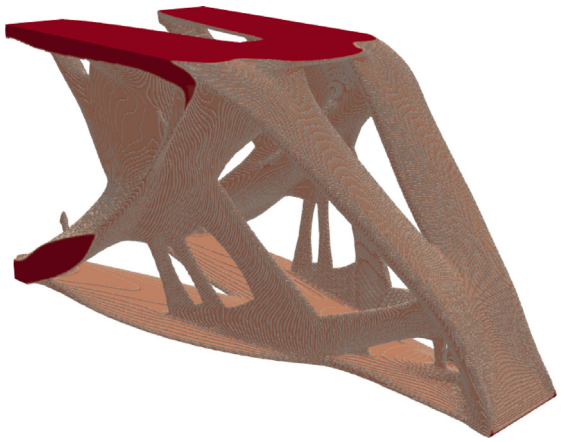


Fig. 4. Linear cantilever example with 65.5 million elements, volume fraction of $V^* = 0.1$, and filter radius of 2.5 elements. Computed using the Futhark implementation on a Nvidia A100 GPU to run 100 design iterations, in 7380 s, approximately 2 h. The shown result is thresholded such that elements with filtered density lower than 0.5 are removed.

4. Numerical experiments

In order to validate the correctness and performance of the GPU implementations, several numerical examples are presented here. Three examples are considered for both linear and nonlinear elasticity, a cantilever (aspect ratio $2 \times 1 \times 1$), the MBB beam (aspect ratio $3 \times 1 \times 1$), and a double clamped beam example (aspect ratio $1.5 \times 1 \times 1$), as shown in Fig. 3. For all presented examples the following material parameters are used $E = 1\,\text{Pa}$ and $\nu = 0.3$.

4.1. Linear elasticity

This section is intended to demonstrate that the presented algorithms produce physically sound designs, and more importantly, to discus the efficiency and scaling of the presented implementations.

Cantilever. The first linear elastic example is the cantilever example. The cantilever domain is shown in Fig. 3(a). A constant traction load is applied on a small circular surface, with radius corresponding to a fifth of the domain width, as shown by green in Fig. 3(a).

A resulting design for the Futhark code is presented in Fig. 4 for a $640 \times 320 \times 320$ mesh with approximately 65.5 million elements. The design was optimised using a volume fraction of $V^* = 0.1$, filter radius $r = 2.5$ elements, and 100 design iterations. The resulting designs for the other two codes are not shown, as they are indistinguishable. The presented result was computed using the Futhark implementation on an A100 GPU in 7380 s ($\sim$ 2 h). In

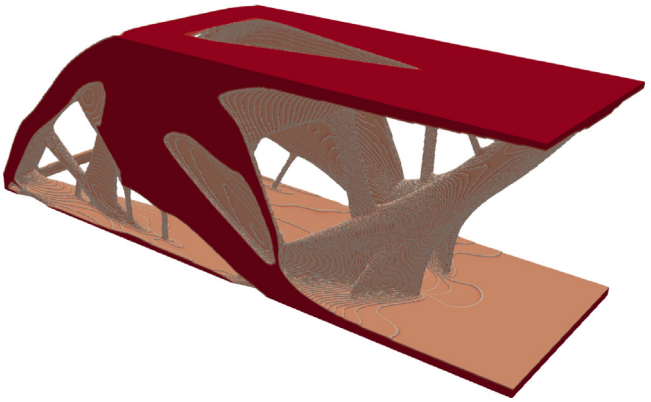


Fig. 5. Linear MBB beam example with 41 million elements, a volume fraction of $V^* = 0.1$, and a filter radius of 2.5 elements. The shown result is thresholded such that elements with filtered density lower than 0.5 are removed.

comparison, the OpenMP implementations are estimated to take approximately 2 h (GPU) and 3.15 h (CPU), by extrapolating the computation time of the first 20 design iterations presented in Fig. 6.

It can be seen that the resulting structure has the expected V-shape near the clamped boundary, as the bending stiffness is increased by placing material near the top and bottom of the domain. The structure has a bottom plate, and a top plate, which curve into two supporting bars for the load.

MBB beam. The classic MBB beam example is also considered. The domain is shown in Fig. 3(a), where the brown line denotes a rolling boundary condition, which allows displacement along the longer aspect ratio, but restricts displacement in to two remaining directions.

Fig. 5 shows a resulting MBB design using the linear formulation. The result is computed using the Futhark implementation, the results from the two other implementations are not shown, as they are identical. This example contains 41 million elements and was computed in 4 h using the Futhark implementation and a Nvidia A100 GPU as accelerator. Compared to the cantilever, the MBB example uses more time to compute for fewer elements. This is due to the fact that the loading and larger domain aspect ratios makes it harder for the iterative solvers, and more iterations are required to solve the system [4].

The resulting structure makes intuitive sense, i.e. it can be seen to have two plates on the top and bottom of the domain with stiffening structure in between. The two round supporting corners are connected by beams, as the rolling support only supports in the vertical direction.

4.2. Performance analysis of linear elasticity

Having established that all the presented implementations are capable of solving the design problem to satisfaction, it is time to consider the scaling and efficiency. To achieve this, the cantilever example is used. The wall-clock time of computing the first 20 design iterations is compared across implementations, for a variety of mesh sizes. The Gnu Compiler Collection (version 11.2) was used to compile the OpenMP-CPU code, the NVIDIA HPC SDK (version 22.5) was used for OpenMP-GPU, and the Futhark compiler (version 0.21.11) with GCC was used for the Futhark code. The used compiler flags can be found in the Makefile of the respective code repositories. It should be noted that the performance of the OpenMP-CPU implementation is sensitive to compiler optimisations, most notably enabling manipulation of floating point expressions assuming associativity greatly improves performance. The OpenCL backend was used for the Futhark compiler. Both the CUDA and OpenCL backends were tested, and it was found that the OpenCL backend resulted in faster GPU-kernels for this specific program. The multicore backend was also tested for the Futhark program, which generates a multithreaded CPU program. However, this backend was found to perform significantly worse than the OpenMP-CPU code. This is somewhat expected, as the main focus of Futhark is compilation to GPU-kernels. For the remainder of this work we consider only the OpenCL backend of Futhark.

Two different test machines are used for the performance benchmarks. For the tests of CPU based implementations a machine with two Intel Xeon 8160 CPUs is used. The same machine also has a Nvidia 1080Ti GPU,

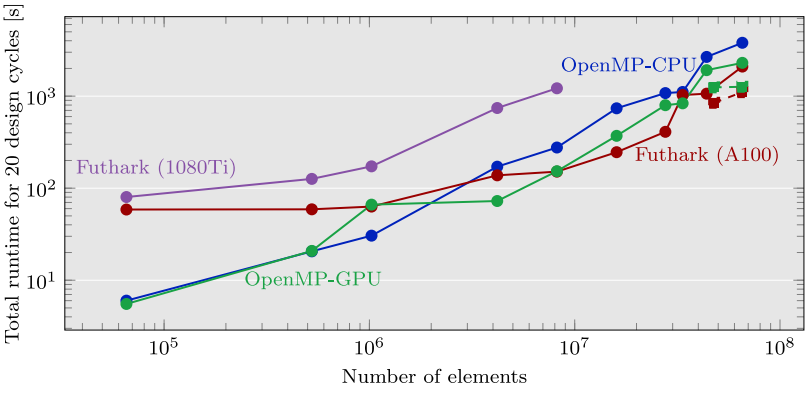


Fig. 6. Runtimes for 20 design iterations of a linear elastic minimum compliance cantilever example with a volume fraction of $V^* = 0.1$. The optimisations were performed using a filter radius of 1.5 elements, resulting in possibly different topologies obtained for every mesh size. The blue and green lines indicate the OpenMP implementations for CPU and GPU respectively. The purple and red lines show the Futhark implementation on two different GPUs. All shown times are computed using 4 multigrid levels, with the exception of the additional points shown with dashed line and square marks, which are computed using 5 levels. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

which is used for GPU acceleration. The other machine used for testing has a Nvidia A100 GPU (80gb), and an Intel Xeon 6226R CPU, which has 16 cores and 700 Gb of RAM. One A100 GPU and the two Intel Xeon 8160 processors have a similar recommended retail price at launch (12,500 USD vs 2× 4700 USD), although the CPUs were launched in 2017, and the A100 in 2020 [25,41]. This is also reflected in their double precision theoretical peak performances, which are 19.5 TFLOPS for the A100 [25], and 3.2 TFLOPS for the two Intel Xeon 8160 processors (2×24 cores $\times$ 32 FLOP/core $\times$ 2.1 GHz). This is an important observation when comparing the performance across various hardware components, in order to achieve a better comparison. Overall, the A100 GPU setup is both slightly more expensive, and is significantly newer than the CPU setup. Ideally, the release date and launch price would match better between the used CPU and GPU configurations, but unfortunately we are limited to the resources available to us. Finally, the CPU benchmark machine is also equipped with a Nvidia 1080Ti GPU, with a theoretical peak performance of 0.35 TFLOPS for double precision, which was launched in 2017 at 699 USD recommended retail price [42]. The 1080Ti is used to show the performance of the GPU implementations on consumer-grade GPU hardware, which is optimised for single precision floating point operations. As such, the 1080Ti is expected to be a bad match for these implementations, which perform key operations in double precision. However, there is a tendency for compute capabilities of high-end GPUs to become available in the consumer grade market after some generations, e.g. tensor cores are becoming available in the consumer cards produced by NVIDIA.

While computing times are in general a stochastic parameter, it was found that the variations in wall-clock time were much smaller than the differences between implementations, therefore results for a single sample are presented.

Two studies are performed, one with varying filter radius and the second with a physically constant filter radius. For the variable filter example, a filter radius of 1.5 elements is used for all tests, meaning that the physical filter radius is decreased when the mesh is refined. This means that the resulting structure is different for every refinement considered. This will affect the iterative solver, which can potentially have great variation in the number of solver iterations across designs. The constant filter radius example uses a filter radius of a 20th of the domain width, corresponding to 1.6 elements on the coarse mesh to 8 elements on the fine mesh.

The timings for the variable filter can be seen in Fig. 6. It can be seen that all implementations are capable of solving large problems in a reasonable time. The slowest implementation is the OpenMP-CPU implementation, solving the problem with 65.5 million elements. It still performs 20 design iterations in 3800 seconds (63 min), which can be extrapolated to about 3.15 h for the entire topology optimisation problem. In contrast, the OpenMP code which is ported to the GPU completes the same 20 design iterations in 2300 s (40 min) using the A100 GPU, which extrapolates to about 2 h to complete the full problem. Similarly, the Futhark implementation completes the 20 design iterations in 2096 s ($\sim$ 35 min). Problems with less that 4 million elements are solved faster by the pure OpenMP-CPU implementation, as the overhead of offloading outweighs the performance improvements.

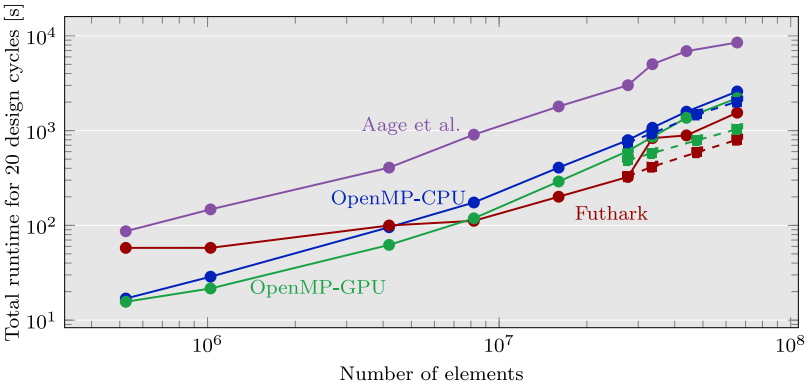


Fig. 7. Runtimes for 20 design iterations of a linear elastic minimum compliance cantilever example with a volume fraction of $V^* = 0.1$. The optimisations were performed using a constant filter radius of 1/20 of the domain width. The blue and green lines indicate the OpenMP implementations for CPU and GPU respectively. The red line shows the Futhark implementation on the A100 GPU. All shown times are computed using 4 multigrid levels, with the exception of the additional points shown with dashed line and square marks, which are computed using 5 levels. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

The two GPU accelerated implementations perform better for the largest problems if a fifth multigrid level is added. These data points are shown as squares in Fig. 6, and the computation times are reduces to 1105 s and 1264 s for Futhark and OpenMP-GPU respectively, reducing the estimated times for the entire optimisation problem to 55 and 63 min respectively. We speculate that the mechanisms for performance improvement are quite different in the GPU implementations. For OpenMP-GPU, the amount of data to be transferred between the GPU and CPU is reduced, along with the size of the direct system to be solved on the CPU. This is beneficial, even at the cost of a slight decrease in the accuracy of the multigrid approximation. In the Futhark implementation however, the accuracy of the coarse grid correction increases, as the fixed number of iterations on the coarse grid can reduce the error further.

Fig. 6 also shows the performance when using the Nvidia 1080Ti GPU with the Futhark implementation. The 1080Ti is representative of a consumer grade GPU, which is more limited in both memory and compute power compared to the A100 GPU. The curve stops at 8 million elements, as the Futhark implementation goes out-of-memory for more finely refined meshes. It is seen that the Futhark implementation running on a 1080Ti is notably slower than the other considered implementations. This is a reflection of the fact that the 1080Ti is optimised towards single precision floating point operations, while our implementations rely heavily on double precision. It is also influenced by the fact that the retail price of the 1080Ti is an order of magnitude lower than the other used compute devices.

The performance comparison for the fixed filter size, as seen in Fig. 7, results in less variation across mesh refinements, as the considered optimisation problem is constant. However, the cost of applying the filter grows with the filter radius, to a point where applying the filters and updating the densities account for approximately 10% of the runtime, for the Futhark implementation with 65.5 million elements, where the filter radius corresponds to 8 elements.

For the comparison with fixed filter size, we have also timed the implementation from Aage et al. [4] based on MPI and PETSc. This test is performed on the same machine as the OpenMP-CPU implementation. It should be noted that this implementation is designed for running on multiple compute nodes on a cluster, although this capability is not used for the present comparison. The problem solved is slightly different, as the PDE-based filter is used. The reason for this difference is that the PETSc implementation assembles the filtering matrix explicitly in memory, resulting in too high computation times and memory usage for large filter radii. By choosing the PDE-based filter, the practical difference for the timing results is minimised, as most of the reported time is spent solving the linear elastic equations. We have changed the solver settings on the coarsest grid from the default SOR preconditioned GMRES, to a direct solution by LU factorisation, to better match the used solver settings used in the other implementations. This change in solver settings improved the performance of the PETSc based implementation for all considered mesh sizes.

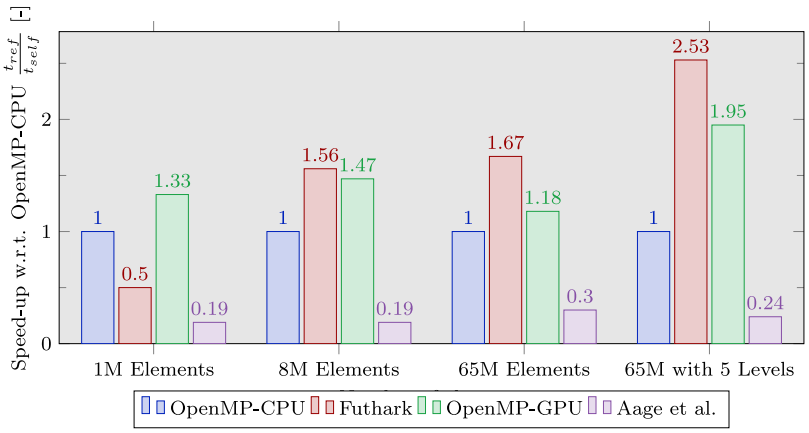


Fig. 8. Relative runtime improvement over the OpenMP-CPU implementation for the first 20 iterations using 4 multigrid levels with constant filter radius. The three mesh refinements correspond to three vertical slices of Fig. 7. Higher is better.

It can be seen that the PETSc based implementation is overall slower than the presented OpenMP-CPU, when running on a single desktop machine. This is expected, as the implementation is designed to solve relatively small systems at every MPI rank, scaling with more MPI ranks for larger problems, e.g. by explicitly assembling all matrices.

For both computational experiments it can be seen that the Futhark implementation has an increase in used runtime for 33M elements and above, most notably in the variable filter radius case. This is likely due to a change of used GPU kernel. Futhark compiles several kernels, and chooses which to launch at runtime based on the input size.

A relative performance comparison for the fixed filter size is shown in Fig. 8. The wall-clock time of the OpenMP-CPU implementation is scaled with the respective wall-clock times, to give an indication of the improvement. We chose OpenMP-CPU as the reference, since it is the fastest implementation without GPU-acceleration.

We see that the OpenMP-GPU implementation is fastest in the coarse case, while the Futhark implementation is significantly slower. Inspecting Fig. 7 it becomes clear that the Futhark implementation is almost unaffected in the runtime until 8 million elements. This indicates an overhead in launching the GPU kernels, which heavily affects the total compute time for small grids. As the OpenMP based implementations are able to complete in less time than the overhead, they outperform the Futhark implementation drastically for coarse problems.

Both GPU kernels perform well at the intermediate refinement of 8 million elements, while they suffer a slowing down in the finest meshes with 4 multigrid levels. As already discussed, we see that adding a fifth multigrid level improves the relative performance of the GPU accelerated implementations drastically due to different reasons. We note that adding a fifth multigrid level only slightly improves the runtime of OpenMP-CPU, as seen by the blue dashed lines with square marks in Fig. 7. The large difference in relative performance when using 4 or 5 multigrid levels for 65M elements is thus explained by a small improvement in runtime for the OpenMP-CPU implementation, along with a drastic drop in runtime for both GPU accelerated implementations.

Interpreting the difference in total runtimes of the presented implementations is not straightforward, as the used multigrid preconditioners also vary. Some performance improvement is due to better utilisation of GPU hardware, some is due to a better suited preconditioner. In order to aid the comparison, Fig. 9 shows the total amount of conjugate gradient iterations spent by the various implementations. The two OpenMP implementations use the same preconditioner, and can thus be compared directly, resulting in a improvement of a factor of 2–4 by using GPU acceleration. Fig. 9 also shows very good agreement between the number of iterations used for these implementations. A small variations occurs between the two implementations, which can be attributed to the difference in floating point execution on the hardware. The Futhark implementation is able to achieve slightly better performance for large problems, even though it uses a coarse grid correction which results in more conjugate gradient iterations, as seen in Fig. 9. Finally, it can be noted that the PETSc based implementation uses very

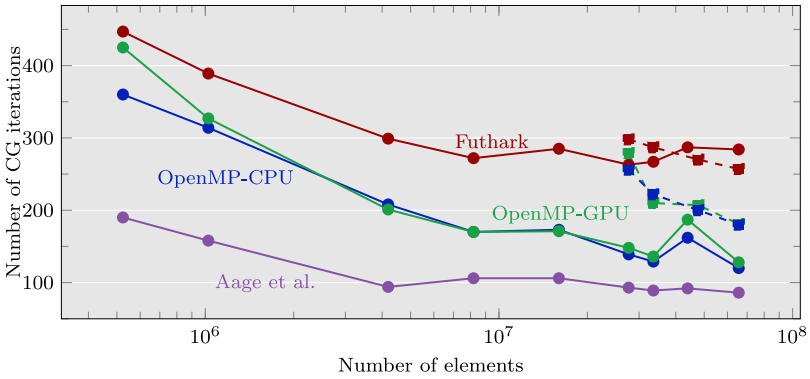


Fig. 9. Sum of used iterations in the conjugate gradient solver for the first 20 design iterations. All shown times are computed using 4 multigrid levels, with the exception of the additional points shown with dashed line and square marks, which are computed using 5 levels.

few conjugate gradient iterations, even though it is the slowest implementation. This is due to the difference of implementation in the smoothing operations, which for the PETSc implementation reduce the error very effectively, but are not possible to implement efficiently for GPU accelerators. We can conclude that both presented approaches are viable for writing GPU accelerated topology optimisation implementations, specially considering the relative ease of implementation.

When increasing the number of multigrid levels, the number of solver iterations increase for the OpenMP based implementations, while remaining somewhat stationary for the Futhark based implementation. Still, the performance improves for both GPU accelerated implementations. In the case of the OpenMP-GPU implementation the amount of data which needs to be transferred between GPU and main memory every iteration is drastically reduced, which could explain the much improved relative performance. Also, the coarse space correction which is performed on the CPU is reduced in complexity as the number of levels increase. Hence, a larger fraction of computing is performed on the GPU. For the Futhark implementation, this decrease of computation time is likely due to a reduction in the computational effort required to solve the coarse grid correction on the GPU.

We note that the performances of the presented GPU codes shown in Figs. 6 and 7 are not necessarily the fastest available. Some other works exploit the structure of topology optimisation problems to achieve high performance with multigrid preconditioning, e.g. Wu et al. [19] and Martínez-Frutos et al. [20]. These works also use lower-level implementations with better control of memory placement and other factors which greatly affect performance. However, as these codes are not publicly available it is not possible to quantify the performance gap between these and the presented implementations.

Memory usage. GPU memory is a very limited and costly resource. Most current cards have memory in the range of 8–16 GiB, while only few expensive cards designed for GPGPU come with more memory. One example of this is the Futhark implementation tested on a 1080Ti card with 8GiB, shown in Fig. 6. The implementation runs out of memory when run with more than 8 million elements, which is why the curve stops earlier than the others.

The GPU memory usage of the two GPU implementations is shown in Fig. 10. Here we see a clear difference between the two approaches, although they both grow linearly in the amount of memory used. The Futhark implementation uses significantly more memory across all mesh sizes. This is a consequence of the incremental flattening strategy implemented by the compiler to automatically transform nested parallelism. This memory usage is the reason that 65 million elements is the limit for the Futhark implementation when using the A100 GPU. The OpenMP-GPU implementation has much lower memory usage, enabling it to run on cards with less memory, and solve larger problems. Here, the main bottleneck is transferring the coarse solution to the CPU every V-cycle. While the Futhark implementation uses much more memory than the OpenMP-GPU implementation, we note that it is still able to fit problems with 65 million elements onto a single GPU, which is sufficient for many use-cases.

5. Extension to nonlinear elasticity

To demonstrate the relative ease with which the presented GPU frameworks can be extended beyond the trivial linear elasticity problems, this sections discuss how to incorporate nonlinearity. Remark that there is only one

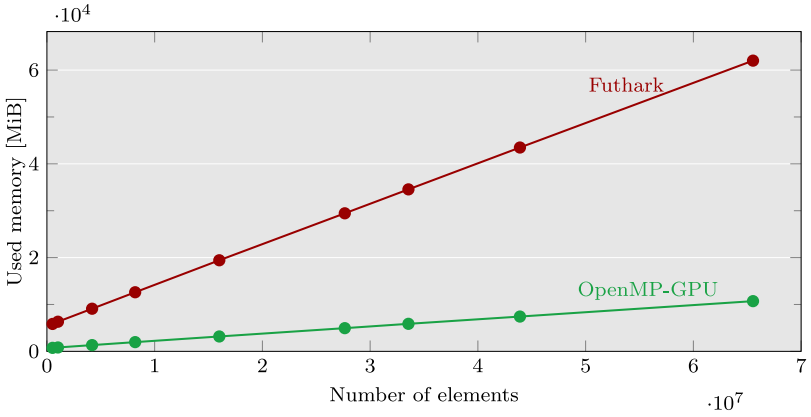


Fig. 10. GPU memory usage of the Futhark and OpenMP-GPU implementations as a function of mesh refinement.

nonlinear elastic implementation included, which is based on Futhark due to ease of implementation. The interested reader may find the implementation at doi:10.5281/zenodo.7791869.

Nonlinear matrix-free approach. It should be noted that the nonlinearity of the problem requires that the local matrices are numerically integrated for every matrix–vector product, as the tangent matrix is not stored in memory. This results in a large increase in computational work compared to the linear case, where a pre-computed local matrix was used to enable fast matrix vector products. Due to this change, an element-wise strategy was adopted for the matrix–vector product, to ensure that every local element matrix needs only to be computed once. In this matrix-free approach all element contributions are computed independently, and the finite element assembly of the nodal values are computed using a generalised histogram implemented in the Futhark core language [43]. That is, instead of using a formulation as seen in Eq. (11), the assembly follows

$$f^e = E_e K^e u^e \tag{13}$$

for every element, followed by a local-to-global assembly of the resulting element forces f^e.

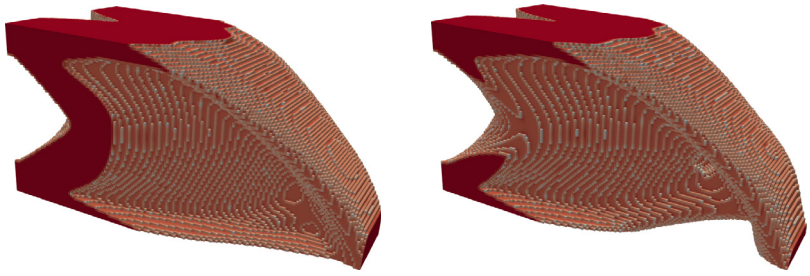
As an example, the Futhark linear implementation takes 1.316ms for a single matrix–vector product, while the nonlinear case takes 1075.4ms, using 2 million elements on the A100 GPU. While this may seem discouraging, it is still possible to perform topology optimisation using this approach, although the presented topology optimisation examples presented in this work do not go beyond 1 million elements.

Newtons method. A Newton–Krylov method with a backtracking line-search is employed for the nonlinear problem [38]. A sequence of linearisations are solved, where each linearisation is approximately solved using the presented multigrid-preconditioned conjugate gradient method. The Newton method is built on top of the modified linear solver in Futhark. Due to the ease of composition in the functional language the implementation of the Newton method itself is very compact.

Load continuation for the nonlinear problem. The initial design iterations of the nonlinear problem can sometimes experience convergence issues, due to a low stiffness for the entire structure compared to the magnitude of the load. In order to resolve this, several approaches are possible. One is to create a continuation on the stiffness penalisation parameter p discussed in Section 2.2, beginning with a linear interpolation, and incrementally increasing the value of p to finally obtain the desired penalisation. Another approach, which was used in this work, is to use a reduced load for the initial design iterations, linearly increasing the load over the first 20 design iterations, until the desired load is reached at iteration 20. Here the calling C code modifies the force vector passed into the Futhark methods, in order to implement this load continuation.

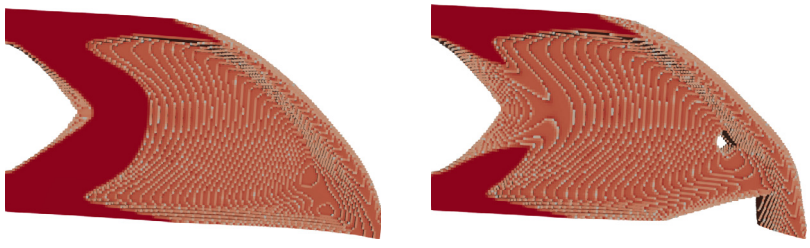
Cantilever. The cantilever example is revisited for the nonlinear elastic problem. The example is computed on a domain of 3 m × 1 m × 1 m domain for a material with $E = 1$ Pa and $\nu = 0.3$. A load of 0.002 N is applied to the cantilever.

From Fig. 11 it can be seen that the structure found using the nonlinear formulation is quite different from its linear counterpart. The nonlinear cantilever does not have a plate on the bottom connecting the support to the loading, as the structure is able to use the reorientation of the hanging beam to decrease the end-compliance. This



(a) Linear. (b) Nonlinear.

Fig. 11. Cantilever computed with 1 million elements, a volume fraction of 0.3, a filter radius of 1.5 elements, and a load of 0.002 N.



(a) Linear. (b) Nonlinear.

Fig. 12. Deformed configurations of the designs from Fig. 11, computed with the nonlinear elasticity. Elements with density below 0.5 are removed for visualisation.

Table 1
Comparison of compliance values for the two designs shown in Fig. 11, under both linear and nonlinear elasticity.

	Linear compliance	Nonlinear compliance
Linear design	$\mathbf{3.82 \times 10^{-4}\,N\,m}$	$3.62 \times 10^{-4}\,\mathrm{N\,m}$
Nonlinear design	$4.13 \times 10^{-4}\,\mathrm{N\,m}$	$\mathbf{3.62 \times 10^{-4}\,N\,m}$

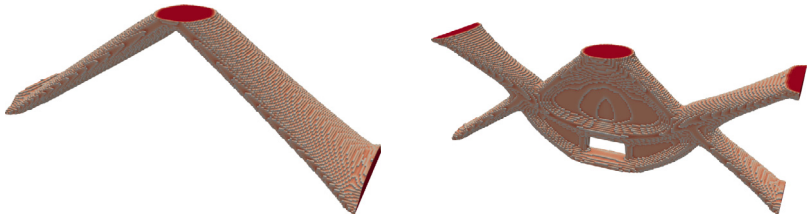
is consistent with the findings of Buhl et al. [26], where similar hanging features are found. The nonlinear structure took approximately 14 h to compute, which can be compared to its linear counterpart, which takes minutes to complete.

Fig. 12 shows the deformed configurations of the cantilever designs, in the nonlinear formulation. It can be seen that for the nonlinear design the beam connecting the load reorients, such that it is in tension under loading. This effectively shortens the cantilever, and allows for a lower end-compliance.

Table 1 shows a cross-check of the linear elastic and end-compliance values for the two cantilever designs. It can be seen that the linear design has a much lower linear compliance compared to the nonlinear design. This is expected due to the bottom plate, which greatly increases the stiffness in the linear regime. The nonlinear end-compliance of the two structures is almost identical, with the nonlinear optimisation being slightly lower, than the linear counterpart (on the 4th decimal).

Double clamped beam. The double clamped beam from Buhl et al. [26] is interpreted in 3D and included here. The design domain shown in Fig. 3(c) shows the modelled domain with a symmetry condition shown in blue. The structure is discretised using $120 \times 80 \times 80$ (768 000) elements. The topology optimisation is performed with a filter radius of 2.5 elements, and a volume fraction of 10%. The nonlinear end-compliance is optimised for a load of 0.00015 newton, on a domain of $1.5 \times 1 \times 1$ metres, using Young's module 1 Pa for the solid.

The linear result is shown in Fig. 13(a), where it can be seen that the linear analysis makes a simple truss structure with two straight bars. While this structure is very stiff for small deformations, it will buckle if the applied load becomes too high. The nonlinear structure shown in Fig. 13(b) is different from the linear structure, as it has two bars connecting to the support on either side of the loaded surface. The upper bars stabilise the structure in regards



(a) Linear. (b) Nonlinear.

Fig. 13. Double clamped beam example computed with 768 000 elements, a volume fraction of 0.1, a filter radius of 2.5 elements, and a load of 0.00015N.

to buckling as they are loaded in tension and do not buckle. The central part of the structure has a vertical plate underneath the loaded surface, which is supported by the trusses where the structure hangs from. The upper part of the structure is similar to the two dimensional example provided in Buhl et al. [26], while the lower structure is similar to the linear result.

6. Discussion

The three implementations for compliance minimisation of linear elasticity show that it is possible to write simple but efficient GPU codes without resorting to explicitly allocating memory and writing separate kernels, as done in e.g. CUDA. The slowest implementation, using OpenMP without GPU acceleration, is able to solve optimisation problems with 65 million elements in approximately 3.15 h on a desktop machine. This is already a large-scale problem, of a scale that usually requires a high performance computing cluster. The GPU accelerated implementations both cut this time to 2 h when using a single Nvidia A100 GPU. This shows that efficient large-scale topology optimisation is not restricted to highly tuned implementations, and can be obtained with high levels of abstraction.

It is especially noteworthy that the Futhark based implementation performs as fast as it does, even though it only employs an approximate solver for the coarse space correction. The Futhark language allows for a concise implementation, which is in contrast to the C implementation where an efficient implementation is seldom the simplest. For instance the overhead of manually tracking indexation between grids accounts for a significant additional amount of code and complexity in the C implementations. This is in large part thanks to the work done on the Futhark compiler, which to our experience emits efficient GPU kernels, with little tuning to the Futhark code.

The implementations rely heavily on the structured nature of the grid. Specifically, the local element matrices are all identical to some scaling, since all elements have identical geometry. This allows for one single element matrix to be integrated offline, and reused for the application of all elements, circumventing the need for numerical integration when applying the stiffness matrix. It might be possible to rewrite the linear implementations for unstructured grids, but not without a significant drop in performance. Another approach to handle arbitrary design domains could be to mesh the entire bounding box, and include passive void domains to restrict the design to the desired domain, as done in Aage et al. [3]. While it might seem counter-intuitive, it might be best for performance to model void elements around a complex domain. The performance improvements obtained from the simplified indexing and similar element matrices, could outweigh the cost of including passive void elements.

Other works such as Wu et al. [19] and Liu et al. [7] both use a structured mesh, but avoid working with the passive void elements arising from embedding a non-trivial design domain. As further work, the presented codes can be extended to enable non-trivial domains by allowing to remove passive void elements. Another option is to include a classic hard-kill strategy, where void elements are removed throughout the optimisation iterations. However, one should take care with updating the used elements, as indexing and non-trivial memory access very quickly becomes prohibitively costly on GPU hardware, potentially negating the performance improvements made by avoiding computation on some elements.

It has also been shown that it is possible to solve nonlinear problems with a million degrees of freedom on the GPU in a reasonable time-frame, although some work is needed before large-scale applications become possible. The nonlinear formulation requires numerical integration of all elements every time the stiffness matrix is needed.

This makes the used matrix-free approach more computationally costly, compared to the linear problem, as the elements need to be numerically integrated every time the tangent matrix is applied. One possible alleviation is to explicitly assemble the full tangent matrix for the nonlinear problem, although this would greatly increase the memory consumption of the program. This would be feasible for the examples considered in this article, as they are no larger than 1 million elements, but could become infeasible for larger meshes as they would no longer fit in the GPU memory.

To summarise, GPU acceleration is now at the point where it is feasible to solve large-scale linear elastic topology optimisation problems on a single desktop system. High level languages and compilers like Futhark simplify the implementation process even further. It is our hope that the provided codes may serve as a basis for future research in topology optimisation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Futhark implementation
OpenMP-GPU implementation
OpenMP-CPU implementation
Futhark Nonlinear implementation

Acknowledgements

The authors would like to acknowledge support from the Villum Foundation, Denmark through the Villum Investigator Project InnoTop.

This project has further received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 951732. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Germany, Bulgaria, Austria, Croatia, Cyprus, Czech Republic, Denmark, Estonia, Finland, Greece, Hungary, Ireland, Italy, Lithuania, Latvia, Poland, Portugal, Romania, Slovenia, Spain, Sweden, United Kingdom, France, Netherlands, Belgium, Luxembourg, Slovakia, Norway, Switzerland, Turkey, Republic of North Macedonia, Iceland, Montenegro.

Appendix. Nonlinear element formulation

This appendix aims to describe the implemented nonlinear formulation thoroughly.

A.1. Basic definitions

Given the undeformed configuration X, and some deformed configuration x, they can be related through a deformation u.

$$x = X + u, \tag{A.1}$$

From this we can find the deformation gradient f_{ij}.

$$f_{ij} = \frac{\partial x_i}{\partial X_j} = \delta_{ij} + \frac{\partial u_i}{\partial X_j}, \tag{A.2}$$

And the Cauchy–Green tensor

$$C_{ij} = f_{ik} f_{kj}, \tag{A.3}$$

From this the Green–Lagrange strain tensor is defined

$$\epsilon_{ij} = \frac{1}{2}(C_{ij} - \delta_{ij}). \tag{A.4}$$

A.2. Neo-Hookean material model

The used Neo-Hookean energy function is

$$\Pi_{int} = \frac{\lambda}{2}(\ln J)^2 + \frac{\mu}{2}(C_{ii} - 3) - \mu \ln J, \quad J = |f_{ij}| \tag{A.5}$$

note that J denotes the determinant of the deformation gradient.

Here the lame parameters are used

$$\lambda = \frac{\nu E}{(1+\nu)(1-2\nu)}, \quad \mu = \frac{E}{2(1+\nu)}, \tag{A.6}$$

The second Piola–Kirchhoff stress for the Neo-Hookean energy is

$$\sigma_{ij} = \frac{\partial \Pi_{int}}{\partial \epsilon_{ij}} = \lambda \ln J C^{-1}_{ij} + \mu(\delta_{ij} - C^{-1}_{ij}), \tag{A.7}$$

And the elasticity tensor is

$$C_{ijkl} = \frac{\partial \Pi_{int}}{\partial \epsilon_{ij} \partial \epsilon_{kl}} = \lambda C^{-1}_{ij} C^{-1}_{kl} + (\mu - \lambda \ln J)(C^{-1}_{ik} C^{-1}_{jl} + C^{-1}_{il} C^{-1}_{jk}). \tag{A.8}$$

A.3. Finite element discretisation of Neo-Hookean material model

Assuming basic FE knowledge and Cook-like notation.

$$[D]_{ref} = \begin{bmatrix} \frac{\partial u^1}{\partial \xi^1} & \frac{\partial u^2}{\partial \xi^1} & \frac{\partial u^3}{\partial \xi^1} \\ \frac{\partial u^1}{\partial \xi^2} & \frac{\partial u^2}{\partial \xi^2} & \frac{\partial u^3}{\partial \xi^2} \\ \frac{\partial u^1}{\partial \xi^3} & \frac{\partial u^2}{\partial \xi^3} & \frac{\partial u^3}{\partial \xi^3} \end{bmatrix} = \begin{bmatrix} \sum_i N_{i,\xi^1} u^1_i & \sum_i N_{i,\xi^1} u^2_i & \sum_i N_{i,\xi^1} u^3_i \\ \sum_i N_{i,\xi^2} u^1_i & \sum_i N_{i,\xi^2} u^2_i & \sum_i N_{i,\xi^2} u^3_i \\ \sum_i N_{i,\xi^3} u^1_i & \sum_i N_{i,\xi^3} u^2_i & \sum_i N_{i,\xi^3} u^3_i \end{bmatrix} \tag{A.9}$$

$$[D] = \begin{bmatrix} \frac{\partial u^1}{\partial x^1} & \frac{\partial u^2}{\partial x^1} & \frac{\partial u^3}{\partial x^1} \\ \frac{\partial u^1}{\partial x^2} & \frac{\partial u^2}{\partial x^2} & \frac{\partial u^3}{\partial x^2} \\ \frac{\partial u^1}{\partial x^3} & \frac{\partial u^2}{\partial x^3} & \frac{\partial u^3}{\partial x^3} \end{bmatrix} = [J]^{-1}[D]_{ref} \tag{A.10}$$

$$[F] = [D] + [I], \quad J = |[F]| \tag{A.11}$$

$$[C] = [F]^\top [F] \tag{A.12}$$

$$[\epsilon] = \frac{1}{2}([C] - [I]) \tag{A.13}$$

$$[\sigma] = \lambda \ln |[F]| [C]^{-1} + \mu([I] - [C]^{-1}) \tag{A.14}$$

The elasticity tensor is renamed E, to avoid clash with the Cauchy–Green deformation tensor.

$$[E] = C_{ijkl} \quad \text{transformed to voigt notation} \tag{A.15}$$

A.4. Finite element discretisation of strains

The following write-up of the Green–Lagrange strains is based on Zienkiewicz and Taylor [44]

The Green–Lagrange strains can be written as

$$\{\epsilon\} = \{\epsilon_0\} + \{\epsilon_L\} \tag{A.16}$$

where $\{\epsilon_0\}$ are the usual linear strains

$$\{\epsilon_0\} = [B_0]\{u\}, \tag{A.17}$$

And $\{\epsilon_L\}$ denote the additional Lagrangian strains,

$$\{\epsilon_L\} = \frac{1}{2}[A]\{\Theta\} = \frac{1}{2}\begin{bmatrix} \{\Theta_x\}^\top & 0 & 0 \\ 0 & \{\Theta_y\}^\top & 0 \\ 0 & 0 & \{\Theta_z\}^\top \\ 0 & \{\Theta_z\}^\top & \{\Theta_y\}^\top \\ \{\Theta_z\}^\top & 0 & \{\Theta_x\}^\top \\ \{\Theta_y\}^\top & \{\Theta_x\}^\top & 0 \end{bmatrix} \begin{Bmatrix} \{\Theta_x\} \\ \{\Theta_y\} \\ \{\Theta_z\} \end{Bmatrix} \tag{A.18}$$

here $\{\Theta_x\}^\top = \{\frac{\partial u}{\partial x}, \frac{\partial v}{\partial x}, \frac{\partial w}{\partial x}\}$, $\{\Theta_y\}^\top = \{\frac{\partial u}{\partial y}, \frac{\partial v}{\partial y}, \frac{\partial w}{\partial y}\}$, and $\{\Theta_z\}^\top = \{\frac{\partial u}{\partial z}, \frac{\partial v}{\partial z}, \frac{\partial w}{\partial z}\}$. Which can be computed similarly to Eq. (A.10).

Similarly, the Lagrange interpolation can be found by

$$[B_L] = [A][G] = [A]\,[[g_1][g_2][g_3][g_4][g_5][g_6][g_7][g_8]] \tag{A.19}$$

where

$$[g_i] = \begin{bmatrix} N_{i,x} & 0 & 0 \\ 0 & N_{i,x} & 0 \\ 0 & 0 & N_{i,x} \\ N_{i,y} & 0 & 0 \\ 0 & N_{i,y} & 0 \\ 0 & 0 & N_{i,y} \\ N_{i,z} & 0 & 0 \\ 0 & N_{i,z} & 0 \\ 0 & 0 & N_{i,z} \end{bmatrix} \tag{A.20}$$

Now the full interpolation is defined as

$$[\bar{B}] = [B_L] + [B_0] \tag{A.21}$$

Finally, the element residual and tangent matrix can be found as:

$$\{R_e\} = \int_{V^e} [\bar{B}]\{\sigma\}dV - \{P_e\} \tag{A.22}$$

$$[k^e] = \int_{V^e} [G]^\top [M][G]dV + \int_{V^e} [\bar{B}]^\top [C][\bar{B}]dV \tag{A.23}$$

where

$$[M] = \begin{bmatrix} \sigma_{11}[I_3] & \sigma_{12}[I_3] & \sigma_{13}[I_3] \\ \sigma_{12}[I_3] & \sigma_{22}[I_3] & \sigma_{23}[I_3] \\ \sigma_{13}[I_3] & \sigma_{23}[I_3] & \sigma_{33}[I_3] \end{bmatrix} \tag{A.24}$$

References

[1] M. Bendsøe, O. Sigmund, Topology Optimization: Theory, Methods, and Applications, Springer Berlin Heidelberg, 2003.

[2] O. Sigmund, K. Maute, Topology optimization approaches: A comparative review, Struct. Multidiscip. Optim. 48 (6) (2013) 1031–1055, http://dx.doi.org/10.1007/s00158-013-0978-6.

[3] N. Aage, E. Andreassen, B.S. Lazarov, O. Sigmund, Giga-voxel computational morphogenesis for structural design, Nature 550 (7674) (2017) 84–86, http://dx.doi.org/10.1038/nature23911.

[4] N. Aage, E. Andreassen, B.S. Lazarov, Topology optimization using PETSc: An easy-to-use, fully parallel, open source topology optimization framework, Struct. Multidiscip. Optim. 51 (3) (2015) 565–572, http://dx.doi.org/10.1007/s00158-014-1157-0.

[5] Nvidia GeForce RTX 3080 family, 2022, NVIDIA, Nvidia URL https://www.nvidia.com/en-gb/geforce/graphics-cards/30-series/rtx-3080-3080ti/. (Accessed 20 June 2022).

[6] Intel xeon® W-3335 processor specification, 2022, INTEL, Intel URL https://ark.intel.com/content/www/us/en/ark/products/217244/intel-xeon-w3335-processor-24m-cache-up-to-4-00-ghz.html (Accessed 20 June 2022).

[7] H. Liu, Y. Hu, B. Zhu, W. Matusik, E. Sifakis, Narrow-band topology optimization on a sparsely populated grid, ACM Trans. Graph. 37 (6) (2019) 1–14, http://dx.doi.org/10.1145/3272127.3275012.

[8] NVIDIA, P. Vingelmann, F.H. Fitzek, CUDA, release: 10.2.89, 2020, URL https://developer.nvidia.com/cuda-toolkit.

[9] J.E. Stone, D. Gohara, G. Shi, OpenCL: A parallel programming standard for heterogeneous computing systems, Comput. Sci. Eng. 12 (3) (2010) 66–73, http://dx.doi.org/10.1109/MCSE.2010.69.

[10] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, J. McDonald, Parallel programming in OpenMP, Morgan kaufmann, 2001.
[11] R. Farber (Ed.), Parallel Programming with OpenACC, Morgan Kaufmann, Boston, 2017, http://dx.doi.org/10.1016/B978-0-12-410397-9.09988-1.
[12] T. Henriksen, N.G.W. Serup, M. Elsman, F. Henglein, C.E. Oancea, Futhark: Purely functional GPU-programming with nested parallelism and in-place array updates, in: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, in: PLDI 2017, ACM, New York, NY, USA, 2017, pp. 556–571, http://dx.doi.org/10.1145/3062341.3062354.
[13] T. Henriksen, M. Elsman, Towards size-dependent types for array programming, in: Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming, in: ARRAY 2021, Association for Computing Machinery, New York, NY, USA, 2021, pp. 1–14, http://dx.doi.org/10.1145/3460944.3464310.
[14] T. Henriksen, F. Thorøe, M. Elsman, C. Oancea, Incremental flattening for nested data parallelism, in: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, PPoPP '19, ACM, New York, NY, USA, 2019, pp. 53–67, http://dx.doi.org/10.1145/3293883.3295707.
[15] A. Paszke, D. Johnson, D. Duvenaud, D. Vytiniotis, A. Radul, M. Johnson, J. Ragan-Kelley, D. Maclaurin, Getting to the point. Index sets and parallelism-preserving autodiff for pointful array programming, 2021, http://dx.doi.org/10.48550/ARXIV.2104.05372.
[16] M. Steuwer, T. Remmelg, C. Dubach, Lift: A functional data-parallel IR for high-performance GPU code generation, in: Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO '17, IEEE Press, 2017, pp. 74–85.
[17] E. Wadbro, M. Berggren, Megapixel topology optimization on a graphics processing unit, SIAM Rev. 51 (4) (2009) 707–721, http://dx.doi.org/10.1137/070699822.
[18] S. Schmidt, V. Schulz, A 2589 line topology optimization code written for the graphics card, Comput. Vis. Sci. 14 (6) (2011) 249–256, http://dx.doi.org/10.1007/s00791-012-0180-1.
[19] J. Wu, C. Dick, R. Westermann, A system for high-resolution topology optimization, IEEE Trans. Vis. Comput. Graphics 22 (3) (2016) 1195–1208, http://dx.doi.org/10.1109/TVCG.2015.2502588.
[20] J. Martínez-Frutos, P.J. Martínez-Castejón, D. Herrero-Pérez, Efficient topology optimization using GPU computing with multilevel granularity, Adv. Eng. Softw. 106 (2017) 47–62, http://dx.doi.org/10.1016/j.advengsoft.2017.01.009.
[21] J. Liu, Z. Xian, Y. Zhou, T. Nomura, E.M. Dede, B. Zhu, A marker-and-cell method for large-scale flow-based topology optimization on GPU, Struct. Multidiscip. Optim. 65 (4) (2022) 125, http://dx.doi.org/10.1007/s00158-022-03214-z.
[22] T. Zegard, G.H. Paulino, Toward GPU accelerated topology optimization on unstructured meshes, Struct. Multidiscip. Optim. 48 (3) (2013) 473–485, http://dx.doi.org/10.1007/s00158-013-0920-y.
[23] J. Martínez-Frutos, P.J. Martínez-Castejón, D. Herrero-Pérez, Fine-grained GPU implementation of assembly-free iterative solver for finite element problems, Comput. Struct. 157 (2015) 9–18, http://dx.doi.org/10.1016/j.compstruc.2015.05.010.
[24] D. Herrero-Pérez, P.J. Martínez Castejón, Multi-GPU acceleration of large-scale density-based topology optimization, Adv. Eng. Softw. 157–158 (2021) 103006, http://dx.doi.org/10.1016/j.advengsoft.2021.103006.
[25] NVIDIA A100 tensor core GPU, 2022, NVIDIA Nvidia URL https://www.nvidia.com/en-us/data-center/a100/. (Accessed 20 June 2022).
[26] T. Buhl, C. Pedersen, O. Sigmund, Stiffness design of geometrically nonlinear structures using topology optimization, Struct. Multidiscip. Optim. 19 (2) (2000) 93–104, http://dx.doi.org/10.1007/s001580050089.
[27] R.D. Cook, D.S. Malkus, M.E. Plesha, Concepts and Applications of Finite Element Analysis, fourth ed., Wiley, New York, NY, 2001.
[28] A. Klarbring, N. Strömberg, Topology optimization of hyperelastic bodies including non-zero prescribed displacements, Struct. Multidiscip. Optim. 47 (1) (2013) 37–48, http://dx.doi.org/10.1007/s00158-012-0819-z.
[29] F. Wang, B.S. Lazarov, O. Sigmund, J.S. Jensen, Interpolation scheme for fictitious domain techniques and topology optimization of finite strain elastic problems, Comput. Methods Appl. Mech. Engrg. 276 (2014) 453–472, http://dx.doi.org/10.1016/j.cma.2014.03.021.
[30] B. Bourdin, Filters in topology optimization, Internat. J. Numer. Methods Engrg. 50 (9) (2001) 2143–2158, http://dx.doi.org/10.1002/nme.116.
[31] O. Sigmund, A 99 line topology optimization code written in Matlab, Struct. Multidiscip. Optim. 21 (2) (2001) 120–127, http://dx.doi.org/10.1007/s001580050176.
[32] E. Andreassen, A. Clausen, M. Schevenels, B.S. Lazarov, O. Sigmund, Efficient topology optimization in MATLAB using 88 lines of code, Struct. Multidiscip. Optim. 43 (1) (2011) 1–16, http://dx.doi.org/10.1007/s00158-010-0594-7.
[33] M. Baandrup, O. Sigmund, H. Polk, N. Aage, Closing the gap towards super-long suspension bridges using computational morphogenesis, Nature Commun. 11 (1) (2020) 2735, http://dx.doi.org/10.1038/s41467-020-16599-6.
[34] Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd Edition, 2003, p. xviii+528, http://dx.doi.org/10.2113/gsjfr.6.1.30, arXiv:0806.3802,
[35] T.H. Nguyen, G.H. Paulino, J. Song, C.H. Le, A computational paradigm for multiresolution topology optimization (MTOP), Struct. Multidiscip. Optim. 41 (4) (2010) 525–539, http://dx.doi.org/10.1007/s00158-009-0443-8.
[36] J. Wu, C. Dick, R. Westermann, A system for high-resolution topology optimization, IEEE Trans. Vis. Comput. Graphics 22 (3) (2016) 1195–1208, http://dx.doi.org/10.1109/TVCG.2015.2502588.
[37] Y. Chen, T.A. Davis, W.W. Hager, S. Rajamanickam, Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate, ACM Trans. Math. Software 35 (3) (2008) http://dx.doi.org/10.1145/1391989.1391995.
[38] J. Nocedal, S.J. Wright, Numerical Optimization, 2 ed., Springer, New York, NY, USA, 2006.
[39] NVIDIA, NVIDIA CUDA example - Convolution Texture, 2023, https://github.com/NVIDIA/cuda-samples/blob/master/Samples/2_Concepts_and_Techniques/convolutionTexture/convolutionTexture.cu, (Accessed 09 Februay 2023).
[40] NVIDIA, NVIDIA CUDA example - Convolution Separable, 2023, https://github.com/NVIDIA/cuda-samples/blob/master/Samples/2_Concepts_and_Techniques/convolutionSeparable/convolutionSeparable.cu. (Accessed 09 February 2023).
[41] Intel xeon® platinum 8160 processor specification, 2022, INTEL Intel URL https://ark.intel.com/content/www/us/en/ark/products/120501/intel-xeon-platinum-8160-processor-33m-cache-2-10-ghz.html (Accessed 20 June 2022).

[42] Nvidia GeForce RTX 10 family, 2022, NVIDIA Nvidia URL https://www.nvidia.com/en-gb/geforce/10-series/ (Accessed 20 June 2022).
[43] T. Henriksen, S. Hellfritzsch, P. Sadayappan, C. Oancea, Compiling generalized histograms for GPU, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '20, IEEE Press, 2020, pp. 1–14, http://dx.doi.org/10.1109/SC41405.2020.00101.
[44] O.C. Zienkiewicz, R.L. Taylor, The Finite Element Method for Solid and Structural Mechanics, Elsevier, 2005.