A Framework for Dynamically-Loaded Hardware Library (HLL) in FPGA Acceleration

Cardarilli, Gian Carlo; Di Carlo, Leonardo ; Nannarelli, Alberto; Pandolfi, Federico Maria ; Re, Marco

Published in:
Proceedings of the 15th IEEE International Symposium on Signal Processing and Information Technology (ISSPIT 2015)

Link to article, DOI:
10.1109/ISSPIT.2015.7394346

Publication date:
2016

Document Version
Peer reviewed version

A Framework for Dynamically-Loaded Hardware Library (HLL) in FPGA Acceleration

Gian Carlo Cardarilli*, Leonardo Di Carlo*, Alberto Nannarelli†, Federico Maria Pandolfi* and Marco Re*
*University of Rome "Tor Vergata"
Dept. of Electronic Engineering
Via del Politecnico 1, 00133, Rome, Italy
Email: g.cardarilli@uniroma2.it, marco.re@uniroma2.it
†Technical University of Denmark
DTU Compute
Richard Petersens Plads, DK-2800, Kgs. Lyngby
Email: alna@dtu.dk

***Abstract*—Hardware acceleration is often used to address the need for speed and computing power in embedded systems. FPGAs always represented a good solution for HW acceleration and, recently, new SoC platforms extended the flexibility of the FPGAs by combining on a single chip both high-performance CPUs and FPGA fabric.**

The aim of this work is the implementation of hardware accelerators for these new SoCs. The innovative feature of these accelerators is the on-the-fly reconfiguration of the hardware to dynamically adapt the accelerator's functionalities to the current CPU workload. The realization of the accelerators preliminarily requires also the profiling of both the SW (ARM CPU + NEON Units) and HW (FPGA) performance, an evaluation of the partial reconfiguration times and the development of an application-specific IP-cores library.

This paper focuses on the profiling aspect of both the SW and HW implementation of the same operations, using arithmetic routines (BLAS) as the reference point for benchmarking, and presents a comparison of the results in terms of speed, power consumption and resources utilization.

I. Introduction

Nowadays the embedded systems market is always craving for more powerful and faster machines, capable of processing a huge amount of data in the shortest time possible. This need for computing power can be related both to the increase of signal-processing algorithms' complexity and to the growth of the amount of data to process.

A hardware acceleration approach is often used to address this issue. This is done to make the most out of the high parallelism achievable with dedicated hardware structures and to offload the CPU from the computational burden that is related to the execution of these operation in a serial fashion by using an ALU. In this field, FPGAs always represented a good trade-off between flexibility, cost, power consumption and time-to-market. In recent years, newer products extended the concept of flexible platform by combining on a single chip high-performance ARM CPUs (Processing System) and FPGA fabric (Programmable Logic), as in the case of the *Zynq™-7000* SoC [6].

The benefits introduced by this kind of platforms for acceleration purposes are remarkable and this is why the aim of this work is the implementation of hardware accelerators for these new SoCs. The innovative feature of the accelerators to be developed is the on-the-fly reconfiguration of the hardware. The design methodology, in fact, will take advantage of the latest techniques in terms of FPGAs partial reconfiguration to dynamically adapt the accelerator's functionalities to the current CPU workload, allowing the full exploitation of the SoC in terms of performance and flexibility.

The realization of the accelerators requires the preliminary characterization of the *Zynq™* Processing System (i.e. ARM CPU in conjunction with NEON Media Processing Engines) performance and the comparison with a HW acceleration approach. We also need to evaluate the partial reconfiguration times and to develop an application-specific IP-cores library to allow the FPGA to adapt itself to the CPU workload. We decided to rename this library as Hardware Link Library (HLL) because the principle is similar to the Dynamic Link Library but it targets actual hardware modules of the design instead of software modules.

In this paper, we will focus only on the profiling aspect, leaving the partial reconfiguration topic and the development of the IP-cores library for future work.

The paper is structured as follows: in Section 2 we will briefly illustrate the target platform for our implementations while Section 3 will describe the implementation for the Processing System (PS) and the case study on NEON units. The results of NEON units acceleration will be presented in Section 4. Section 5 will deal with the proposed HW implementations and the comparison between HW and SW results (in terms of speed and power consumption) will be presented in section 6 and 7. Finally, we draw the conclusions in Section 8.

II. Implementation Platform

The hardware accelerators are implemented within the Z-7020 device of the *Zynq™-7000* SoC family. The architecture of the *Zynq™* comprises two different parts: the Processing System (PS) and the Programmable Logic (PL). These two

sections are independent (they also have distinct power domains) and can be used separately or in conjunction. The interconnection between the PS, PL and the software-configurable I/O peripherals is provided by the AMBA AXI bus.

A. Processing System

The Processing System (PS) of the device features a dual-core ARM Cortex-A9 with a 32-bit RISC architecture [2]. Along with the processor, a set of processing resources are available within the Application Processing Unit (APU) of the Zynq. The most important resources for this work are the two NEON Media Processing Engines for SIMD operations [1], the 32KB L1 and 512KB L2 caches and the 256KB On-Chip-Memory (OCM) RAM. A multi-protocol DDR memory controller is provided to support external DRAM memories.

It is worth mentioning that the Programmable Logic configuration is managed by the PS.

B. Programmable Logic

The other main portion of the Zynq-7000 SoC is the Programmable Logic (PL). This general purpose 28nm FPGA fabric is based on the Xilinx Artix-7 technology. The key features, other than the Configurable Logic Blocks (CLBs), are the dual-ported 36Kb BlockRAMs (dedicated memory resources) and the DSP48E Slices (dedicated silicon resources for DSP and high-speed arithmetic).

C. Zedboard and Board Power Measurements

In order to work with the *Zynq™* SoC we used the Zedboard development board. This boards features a XC7Z020 Zynq device, 512MB ($2 \times 128Mb \times 16$) DDR3 memory and a comprehensive set of peripherals.

Among the other things, the Zedboard also features a pair of current-sense pin-headers that are used to measure the power consumption of the board. These headers straddle a $10\ m\Omega$, 1%, 1W current sense resistor which is placed in series with the 12 V power supply. The power can be calculated using the following formula:

$$P = \frac{V_m}{10\ m\Omega} \cdot 12V \quad [W] \tag{1}$$

where V_m is the voltage drop (in millivolt) across the resistor.

III. Proposed SW Implementations

As mentioned in the previous sections, the PS of the Zynq™ comprises dedicated architectures for Single Instruction Multiple Data (SIMD) operations. These architectures are named NEON Media Processing Engines (MPE) or NEON Units and can offer a certain amount of parallelism, with some benefits over the standard CPU approach.

Before implementing hardware accelerators we believe that it is of paramount importance to have a comprehensive knowledge of the capabilities and limitations of the ARM CPUs in conjunction with the NEON units, especially when targeting the smaller devices of the Zynq-7000 family, which have a limited amount of PL resources.

A. NEON Units SW acceleration

The basic concept behind NEON's SIMD technique is that the data to be processed is packed into special wide registers that can hold multiple smaller words. In this way, by specifying a single operation over these registers, multiple data values are processed in parallel using just a single instruction, with benefits over the standard Single Instruction Single Data (SISD) approach.

The potential of this methodology is fully exploited when simple and repeated operations have to be performed on large data sets made of elements that have small word-lengths (up to 32 bits).

NEON units can handle both single precision floating-point and signed/unsigned integer data types but not double-precision floating-point.

There are four ways to boost the SW performance with NEON Units:

- Using NEON optimized libraries.
- Using automatic vectorization from the compiler.
- Using NEON intrinsics.
- Optimizing NEON assembler code manually.

In this work we decided to test NEON units' performance using both the automatic vectorization and the intrinsics methodology.

B. BLAS and NEON intriniscs

The tests were carried out targeting some of the operations specified in the Basic Linear Algebra Subprograms, or BLAS, routines [7], [4]. We decided to use BLAS routines as these are low-level routines that represent a standard for basic vector, matrix and linear algebra operations. Moreover, BLAS implementations are often optimized for speed on a particular machine and can take advantage of dedicated floating-point hardware such as vector registers and SIMD architectures, as in the case of NEON units.

Some routines are often used to measure performance. For example, the LINPACK Benchmark, which is a common measure of a system's floating-point performance, relies heavily on the GEMM, a Level 3 BLAS routine.

For these reasons, in this work, we translated in C one function from the Level 1 (vector-vector operation) and one from the Level 3 (matrix-matrix operation) from the original Fortran source code (we will refer to these versions as C-BLAS) and later we optimized the code for the NEON units using NEON intrinsics.

C. Implemented routines

The selected routines for our implementation were SDOT and SGEMM, where the prefix "S-" indicates that the operations will be performed on single-precision floating-point elements.

Although the BLAS routines support multiple data-formats, we decided to use floating-point numbers because benchmarking tests are usually referred to floating-point operations.

Since we are using 32-bit words for floating-point elements of the vector, the maximum parallelism achievable with the

NEON units is 4 if we use the NEON registers in the 128-bit (Q) configuration.

The target routines perform the following operations:

1) SDOT: produces the dot (scalar) product of two vectors:

$$dot \longleftarrow \mathbf{x}^T\mathbf{y} \tag{2}$$

2) SGEMM: performs the multiplication between matrices:

$$C \longleftarrow \alpha \cdot \mathbf{A} \cdot \mathbf{B} + \beta\mathbf{C} \tag{3}$$

IV. NEON Units Acceleration Results

Each routine was tested in both the C-BLAS and in the NEON intrinsic version using five different optimization options [3]:

1) C-BLAS "as-is", without any optimization from the compiler (the Optimization Level option in the GCC compiler settings was set to -O0.)
2) C-BLAS optimized by the compiler using the automatic vectorization option (Optimization Level set to -O1) [10].
3) Same as point 2 but with an Optimization Level of -O2.
4) Same as point 2 but with an Optimization Level of -O3.
5) NEON intrinsic version of the code with an Optimization Level of -O3. This configuration is the fastest and represent the most optimized solution tested.

The timer, available in the PS, is used to measure the execution time of each subroutine. This timer runs at half the CPU clock frequency and has clock period of 3 ns. Therefore, the execution time was calculated as follows:

$$Execution\,time = N.\,clock\,cycles \cdot 3 \quad [ns] \tag{4}$$

The tests are performed both enabling and disabling the L1-L2 caches in order to evaluate the impact of cache optimization over execution time.

A. SDOT results

Fig. 1 shows the curves obtained for the SDOT function. In these figure the -O2 and -O3 curves overlap as the execution times are almost the same.

It is worth noting that there is a considerable speed-up factor between the non-optimized custom version and the version optimized with automatic vectorization. This gap is even bigger if we consider the version optimized with NEON intrinsics. There is not a big difference in execution times, instead, between the various Optimization Levels. The difference between these levels, though, reside in the code size as, increasing the optimization level, the code size increases as well.

The following table shows the average speed-up factors for the various optimizations compared to the non-optimized solution in the case of cache enabled:

The speed-up factor with the automatic vectorization from the compiler is almost four, reflecting the parallelism of the NEON operations. The speed-up obtained with the NEON intrinsic solution almost doubles that value, demonstrating

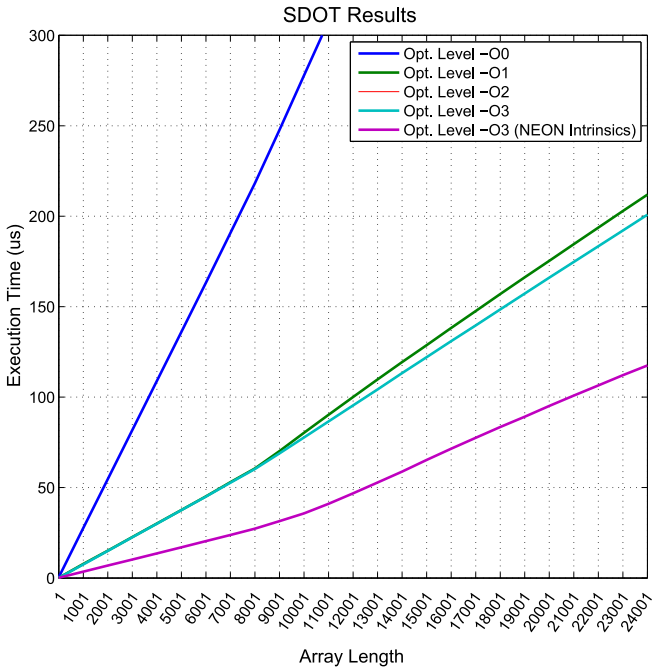

Fig. 1. SDOT results

Optimization Level	Speed-up factor
-O1 (C-BLAS)	$\sim 3,35\times$
-O2 (C-BLAS)	$\sim 3,48\times$
-O3 (C-BLAS)	$\sim 3,51\times$
-O3 (NEON intrinsic)	$\sim 6,63\times$

TABLE I
SDOT Optimization Levels vs. speed-up factors

that an additional improvement can be achieved using a code tailored to the NEON architecture.

Moreover, we can notice that there is a knee in the curves, in particular in NEON intrinsic one, placed in correspondence to arrays length of 8000. One possible explanation is that, being the 32KB L1 and 512KB L2 caches non-inclusive, at this point the L1 hit-rate decrease abruptly and the CPU starts retrieving data from the slower L2 cache. In the cache-disabled case the knee is not visible but the execution times are, by contrast, much higher. Proper caching may, therefore, be a solution to avoid this performance drop.

B. SGEMM results

Fig. 2 shows the results obtained for the SGEMM routine. In these figure, the three Optimization Levels -O1, -O2 and -O3 are not distinguishable as we got almost the same timing results for each case.

It can be noted that there is a discontinuity in the trend of the curves for matrices dimensions of 350×350. This is reflected by some extent also in the speed-up factors as it can be seen in the next table:

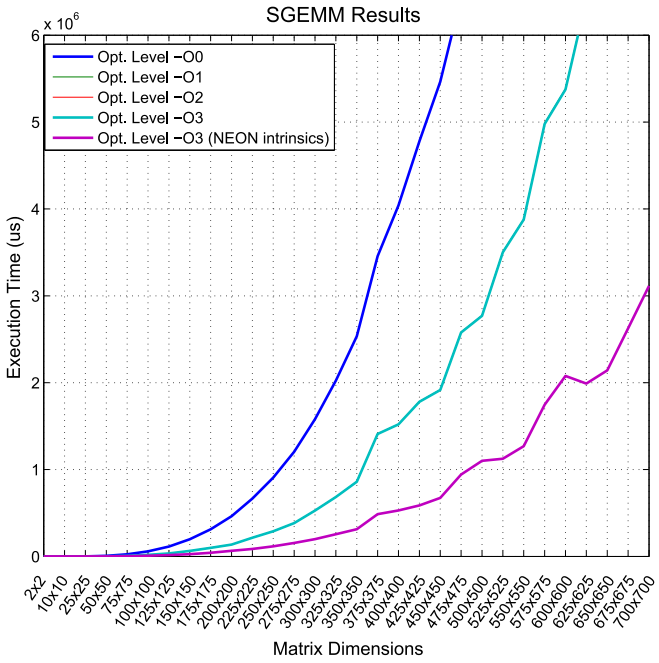

Fig. 2. SGEMM results

Optimization Level	Speed-up factor (before 350×350)	Speed-up factor (after 350×350)
-O1 (C-BLAS)	$\sim 3,42\times$	$\sim 2,66\times$
-O2 (C-BLAS)	$\sim 3,44\times$	$\sim 2,67\times$
-O3 (C-BLAS)	$\sim 3,45\times$	$\sim 2,67\times$
-O3 (NEON intrinsic)	$\sim 6,35\times$	$\sim 7,85\times$

TABLE II
SGEMM OPTIMIZATION LEVELS VS. SPEED-UP FACTORS

This is a cache inefficiency issue since the amount of memory needed to store a matrix of single precision floating-point elements with dimensions of 375×375 is:

$$375 \times 375 \times 32\,bits = 562,5\,KB \tag{5}$$

Since the L1 (32 KB) and L2 (512 KB) caches in the Processing System of the Zynq™ are non-inclusive, they can store in total up to 544 KB of data. Matrices with dimensions greater than 350×350 exceed that capacity, meaning that the data caches hit-rate in those cases may abruptly decrease if memory access is not handled properly, making the whole process more DDR dependent and, hence, slower.

This may also explain why the trends are more regular before 350×350 and suddenly become more jagged after that dimension and also why the discontinuities in the cache-disabled test (in which DDR is used all the time) are not visible (it must be reported, though, that the execution times are much higher in this case).

The tests also show that the speed-up factors obtained for SGEMM function are similar to the SDOT one and that the NEON intrinsic solution is still much faster than the others and that the various optimization levels lead to almost the same results, with minimal differences in speed but with differences in code size.

V. PROPOSED HW IMPLEMENTATIONS

As already hinted, we believe that it is important to have also HW profiling data to compare the two approaches. Therefore, we decided to implement on the PL an architecture that resembled the instructions executed by the NEON units in the SGEMM case.

We took in consideration just the basic $A \times B$ multiplication, with $\alpha = 1$ and $\beta = 0$ and without transposing the matrices.

The architectures implemented execute the following tasks:

- Retrieve the input matrices using an AXI DMA and a 32-bit AXI Stream interface.
- For every element of matrix C perform the dot-products between the rows of matrix A and columns of matrix B using different levels of parallelism.
- Output the resulting matrix using the same DMA but with a different AXI Stream interface.

The HW implementations were realized using the new Vivado HLS tool [13] and the matrices dimensions were fixed to 32×32. In this way each matrix has 1024 floating-point elements. AXI DMA transfers with more than 1024 words cannot be sent with just one transaction and thus require to be split in multiple transfers, increasing the whole communication overhead. Moreover, even with 32×32 matrices, to implement a fully parallel architecture (parallelism = 32) the number of resources required is a considerable percentage of the available resources of the Z-7020 Zynq™ device and the final design occupy almost the entire FPGA (if we take into account the resources needed by the AXI DMA and the interconnections).

Three different solutions were developed to measure the impact of parallelism over the achievable performance. The parallelism values chosen for the three solutions were respectively: 4 (the same parallelism of the NEON architecture), 16 and 32 (the maximum parallelism directly achievable with 32x32 matrices).

Regarding clock frequencies, the PS runs at 667 MHz while the HW synthesized in the PL can sustain a maximum clock frequency of 100 MHz.

VI. HW VS. SW ACCELERATION RESULTS

Fig. 3 shows the execution times in nanoseconds for the various implementations, both HW and SW.

As we can see from the histograms, having the same parallelism on both the PS and PL does not lead to the same results. This is because, even if the number of clock cycles in the custom IP core to output the results is almost half compared to the most optimized NEON intrinsic case and almost one third of the automatic vectorizations cases, the clock in the FPGA is much slower than the CPU's one. Therefore, the execution time is much higher in the PL than in the PS.

With a parallelism of 16 in the custom IP core we get almost the same execution time than in the compiler-optimized cases

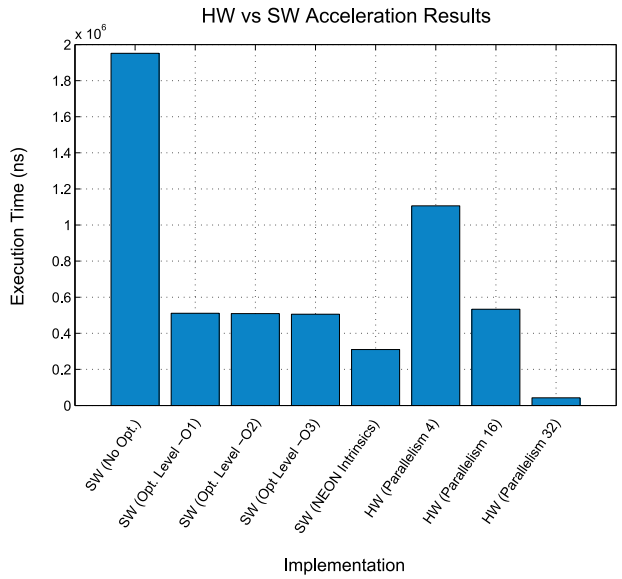

Fig. 3. HW vs. SW Acceleration Results

but this solution is still slow when compared to the NEON intrinsic one.

We estimated that, in order to have the same execution time on both the NEON intrinsic case and in the hardware IP case, a parallelism of at least 26 is needed for the hardware implementation. We based this assumption on the following formula:

$$\begin{matrix}\text{PS}\\ \text{clock}\\ \text{frequency}\end{matrix} \times \begin{matrix}\text{NEON}\\ \text{parallelism}\end{matrix} = \begin{matrix}\text{PL}\\ \text{clock}\\ \text{frequency}\end{matrix} \times \begin{matrix}\text{IP}\\ \text{core}\\ \text{parallelism}\end{matrix} \quad (6)$$

Once the parallelism of the hardware IP core exceeds this theoretical cross-point the hardware architecture starts being faster than the PS implementation.

The speed-up factors for the three different hardware implementations are summarized in the table below (these factors are calculated in relation to the NEON intrinsic case):

Hardware accelerator parallelism	Speed-up factor
4	$\sim 0,28\times$
16	$\sim 0,58\times$
32	$\sim 7,26\times$

TABLE III
HW ACCELERATOR PARALLELISM VS. SPEED-UP FACTORS

As expected, this performance boost comes with a price. The number of resources needed to implement the parallelism 32 solution is indeed very high. The estimated resources utilization from Vivado HLS for this latest hardware IP solution are about 75% of the available DSP Blocks, 23% of BlockRAMs, 45% of LUTs and 12% of flip-flops. Please consider that the additional resources needed to implement the AXI DMA and the AXI Interconnections are not included in this estimation.

To implement a single floating-point multiplier 3 DSP48E blocks are needed while to implement a single floating-point adder additional 2 DSP48E block are employed. If we take that into account we can figure out that implementing floating-point operations on the Programmable Logic may not always be a good solution, also considering that, to match the NEON units performance, a high degree of parallelism is required. In our case we managed to fit in the PL an accelerator faster than the NEON units without effort. This may not be the case for more complex designs to be implemented on the Z-7010 or Z-7020 and, during the partitioning phase of the project, the use of NEON units for floating-point operations instead of an hardware accelerator in the PL is a design choice to consider.

VII. HW VS. SW POWER MEASUREMENTS

The tests were performed targeting three different designs:

- PS + NEON units with intrinsics optimization (without any IP core instantiated in the PL)
- PS + HW IP core with parallelism 4
- PS + HW IP core with parallelism 32

For each design we measured the power consumption of the board during five different stages:

1) When powering the board
2) During the configuration phase
3) After the configuration phase
4) During the execution of the application
5) After the execution of the application

The results obtained are shown in Table IV:

	PS+NEON no PL (W)	PS+PL parallelism 4 (W)	PS+PL parallelism 32 (W)
Power-on	2,349	2,345	2,324
During configuration	2,365	2,358	2,342
After configuration	2,255	2,255	2,240
Application running	3,472	3,467	3,501
After run	3,382	3,432	3,469

TABLE IV
POWER CONSUMPTION

The board power consumption for the three designs is almost the same. This means that the Zynq™ device has a low impact on the overall power consumption and so it is for the different parallelism architectures. Thus, in this case, to compare the different designs, it makes more sense to evaluate the impact on energy consumption rather than on power consumption.

Therefore, by considering the energy consumption, the faster is the implementation, the lower is the execution time and more energy can be saved. In other words, the fastest implementation is also the most power efficient one and this opens up a new frequency trade-off to take into account: raising the operating frequency may slightly increase the overall power consumption but will decrease the execution time, improving power efficiency. This aspect is very important

especially in those cases where many accelerators are employed simultaneously (e.g., data centers) because the energy that can be saved from a single accelerator is multiplied by a large factor.

A final word has to be said about DRAM power consumption. It can be noted that power consumption for the two subsequent idle states (before and after the execution of the application) changes significantly. This is related to the DRAM usage as these memories are responsible for almost one-third of the total run-time and after-run power consumption. All the designs tested use the DDRs available on the Zedboard. These memories are activated and configured when launching the application and remain active even after the application is executed.

Unfortunately, at this moment, it seems not to be possible to dynamically deactivate the DRAM when it is not needed, the only way to do it is to disable it from the beginning, when designing the system within Vivado IDE. This is very inefficient from an energy consumption point of view.

VIII. Conclusion

The aim of our work is to develop efficient accelerators for emerging SoC platforms.

In this preliminary work, we focused on benchmarking different acceleration solutions to understand the implementation tradeoffs necessary to develop a library of hardware IP blocks which are loadable on-demand on the PL.

For this purpose, we selected a few common computation intensive routines and implemented them in the Zynq SoC using several paradigms for acceleration: from no acceleration, to acceleration in the NEON units, to customized IP blocks mapped in the PL.

The experimental results show that, for floating-point operations and low levels of parallelism in the PL, the NEON approach is more convenient. By increasing the level of parallelism, we can get better performance in the PL acceleration, at expenses of higher resources utilization.

Regarding power and energy, one of the main reasons to use hardware acceleration is to improve energy consumption and power efficiency. During our tests we found out that, since the power dissipation values are very similar for the various designs implemented and the Zynq has a low impact over the total board power consumption, the faster implementation is usually also the more power efficient one as it requires less energy to complete the execution.

References

[1] ARM. *Cortex-A9 NEON Media Processing Engine*, version r3p0, 2011.

[2] ARM. *Cortex-A9 Technical Reference Manual*, version r4p1, 2012.

[3] ARM. *NEON Programmers Guide*, version 1.0, 2013.

[4] Netlib BLAS, http://www.netlib.org/blas

[5] D. Bagni, J. Noguera, and F. M. Vallina. *XAPP1170 - Zynq-7000 All Programmable SoC accelerator for floating-point matrix multiplication using Vivado HLS*, version 1.1, 2013.

[6] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart. *The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*. Strathclyde Academic Media, first edition, 2014.

[7] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F.T. Krogh. *Basic Linear Algebra Subprograms for FORTRAN usage*. *ACM Transactions on Mathematical Software*, vol. 5, number 3, pages 308–323, 1979.

[8] S. Rousseaux, D. Hubaux, P. Guisset, J.-D. Legat, *A High Performance FPGA-Based Accelerator for BLAS Library Implementation*. *Proceeding of the Reconfigurable Systems Summer Institute 2007, RSSI 07, July 2007*

[9] Ling Zhuo, V.K. Prasanna, *Design tradeoffs for BLAS operations on reconfigurable hardware*. *in Parallel Processing, 2005. ICPP 2005. International Conference on*, pp.78-86, 14-17 June 2005, doi: 10.1109/ICPP.2005.31

[10] Haoliang Qin. *XAPP1206 - Boost software performance on zynq-7000 AP SoC with NEON*, version 1.1, 2014, http://www.xilinx.com/support/documentation/application_notes/xapp1206-boost-sw-performance-zynq7soc-w-neon.pdf.

[11] Xilinx. *DS190 - Zynq-7000 All Programmable SoC Overview*, version 1.7, 2014, http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

[12] Xilinx. *UG821 - Zynq-7000 All Programmable SoC Software Developers Guide*, version 11.0, 2015, http://www.xilinx.com/support/documentation/user_guides/ug821-zynq-7000-swdev.pdf.

[13] Xilinx. *UG902 - Vivado Design Suite User Guide - High-Level Synthesis*, version 2015.1, 2015, http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug902-vivado-high-level-synthesis.pdf.