Chisel Usecase: Designing General Matrix Multiply for FPGA

Bruno Ferres(✉), Olivier Muller, and Frédéric Rousseau

Univ. Grenoble Alpes, CNRS, Grenoble INP, TIMA, 38000 Grenoble, France
{bruno.ferres,olivier.muller,frederic.rousseau}@univ-grenoble-alpes.fr

Abstract. To ease developers work in an industry where FPGA usage is constantly growing, we propose an alternative methodology for architecture design. Targeting FPGA boards, we aim at comparing implementations on multiple criteria. We implement it as a tool flow based on `Chisel`, taking advantage of high level functionalities to ease circuit design, evolution and reutilization, improving designers productivity.

We target a Xilinx VC709 board and propose a case study on General Matrix Multiply implementation using this flow, which demonstrates its usability with performances comparable to the state of the art, as well as the genericity one can benefit from when designing an application-specific accelerator. We show that we were able to generate, simulate and synthesize 80 different architectures in less than 24 h, allowing different trade-offs to be quickly and easily studied, from the most performant to the less costly, to easily comply with integration constraints.

Keywords: Chisel · FPGA · GEMM · Methodology

1 Introduction

As FPGA usage for application acceleration increases in the industry, notably in the domain of Cloud computing [3,7], RTL based design methodology - *i.e.* the standard methodology in industry - can be questioned on criteria such as efficiency, reusability, or accessibility.

The last decade has witnessed the appearance of new technologies easing hardware development, with higher levels of abstraction. The most known of those are High Level Synthesis, which goal is to bring the power of dedicated hardware acceleration to hardware-agnostic software developers. Nevertheless, HLS still has to cope with some flaws, including fine tuning on the code to infer efficient hardware, as well as lack of control on the generated hardware.

On the other hand, more hardware aware initiatives have appeared in the scientific community, like `Chisel` [2]. `Chisel` (Constructing Hardware In a Scala Embedded Language) is an open source Scala based language dedicated to hardware generation, with high level programming functionalities, and an ever growing community. It can be used to generate Verilog code, insuring compatibility

Grenoble INP—Institute of Engineering Univ. Grenoble Alpes.

F. Rincón et al. (Eds.): ARC 2020, LNCS 12083, pp. 61–72, 2020.
https://doi.org/10.1007/978-3-030-44534-8_5

with the standard flow, and ease design reutilisation thanks to the software constructs it embeds.

Google used Chisel for the design of their Edge TPU [1], and two RISC-V implementations have been proposed - **Rocket Chip** and **BOOM** - showing that the initiative can be integrated in both industrial and academic worlds. Works like [8] showed that Chisel can be used to explore different implementations of a circuit, here designed for BLAS (Basic Linear Algebra Subroutines) dot product acceleration.

BLAS introduces a set of linear algebra operations that can be used to evaluate implementation performances on this kind of applications [5]. In particular, it includes the General Matrix Multiply (GEMM) algorithm, a highly indicative application for all algebra computations [9], which has been deployed to various platforms before, including FPGA [4] and GPU [6]. GEMM is usually implemented with variations on the type and length of elements used, SGEMM and DGEMM respectively representing simple and double precision floating points, and other implementations targeting fixed point or integer representations.

This paper introduces a methodology for designing, testing and evaluating an application using Chisel, demonstrating its usage on a GEMM case study. Through this usecase, we show that our methodology allows to deeply control the generated hardware, producing accelerators that are not only comparable to the state of the art, but behave as they are designed to. Resource usage can be fully explained by targeted architecture, as no hardware inference has to be made by the compile pass. On the other hand, this flow allows to easily explore multiple architectures, studying the influence of application and target parameters on the produced designs. It enables changing the type and width of the operands, the capacities of communication or even the dimension of the applicative problem. Section 2 introduces the aim and steps of this methodology, while Sect. 3 demonstrates its usage on a GEMM usecase. Section 4 presents the results of this usecase, as well as the functionalities of our tool flow, and Sect. 5 discusses the contribution of our work.

2 High Level Methodology

To efficiently implement an application, one must take into account the application temporal behavior, as well as the environment target, in order to take advantage of the available resources. For example, targeting a FPGA requires the developer to consider the different kind of memory and computational resources embedded, the communication links, the reachable clock frequency, and other factors that will impact the choices of implementations.

On the other hand, to maximize the reusability of the developed design, you need to think about genericity before implementation, so generated designs can be adapted to new implementation constraints if needed, including target change.

This section describes the chosen technology for our methodology, as well as its steps, from application and target specificities to implementation.

2.1 High Level Description

To improve productivity when it comes to hardware development, developers must be able to define architectures in a generic way, to be able to generate different implementations by varying application specific and non-specific parameters (*e.g.* I/O size, element type, ...). Such generic implementation would allow to explore different trade-offs, and be able to suggest the most efficient architecture, the most performant or the less costly, for example.

To do so, we choose to use `Chisel` in our architecture generator flow, since the language offers higher level generic features, compared to the ones proposed by standard RTL languages (such as Verilog, SystemVerilog or VHDL).

Replacing RTL. Although `Chisel` remains a RTL language, we identified three main features of it that can ease the development of such generator, compared to standard RTL languages such as VHDL or Verilog.

Table 1 compares `Chisel` and standard RTL features when it comes to parametrized design generation. We can notice that the third feature - high level generation - is also available in both Verilog and VHDL languages, but that the two other features require external tools and complex constructs to be included in these languages.

Table 1. Feature comparison between `Chisel` and standard RTL languages for design generation

Chisel programming feature	RTL equivalent (Verilog and/or VHDL)
Type genericity	Black boxing type specific operators + string replacement in RTL code (*i.e.* using `sed`) *N.B.* `generic` can be used for width genericity only
Procedural programming	Multiple version of the same code to change functionalities and/or behavior
High level generation	`for` or `generate` loops `if` statements

HLS vs `Chisel`. Choosing `Chisel` over standard RTL languages for design implementation can thus be motivated by a need for higher abstraction level when it comes to hardware development.

However, as stated in introduction, High Level Synthesis technology aims at easing accelerators development by synthesizing algorithmic description to hardware circuit, meaning that it is a good candidate for increasing developers productivity, as well as easing design reusability. Yet, HLS requires, by design, inference from the compilation tool to generate functioning circuits. This means that one can not control easily the hardware generated, which allows software or application developers to use it with no hardware knowledge, but also means that experienced hardware developers can not directly control the generation flow, and can only try to tune the code and the tool to orientate the compilation toward an acceptable architecture.

Zhao et al. [11] state that HLS generated circuits can be compared to RTL written ones, but only with hard code discipline and fine pragma tuning. It can be complex, and might need to be repeated each time you change the generation constraints, meaning that evolving your design can be time consuming.

Since we aim at reusing code and generating multiple accelerators with different constraints, we chose to use the open-source, highly promising technology of `Chisel`.

2.2 From Application to Architecture

Using a language like `Chisel`, we propose a more generic development methodology, relying on hardware knowledge about the chosen target family (*e.g.* does it propose external memory, which kind of computing units are available, ...), and an architectural study of the application. This methodology does not require specific adjustment of code for a particular target board, but rather a generic design for a class of target, making it more **target-agnostic**.

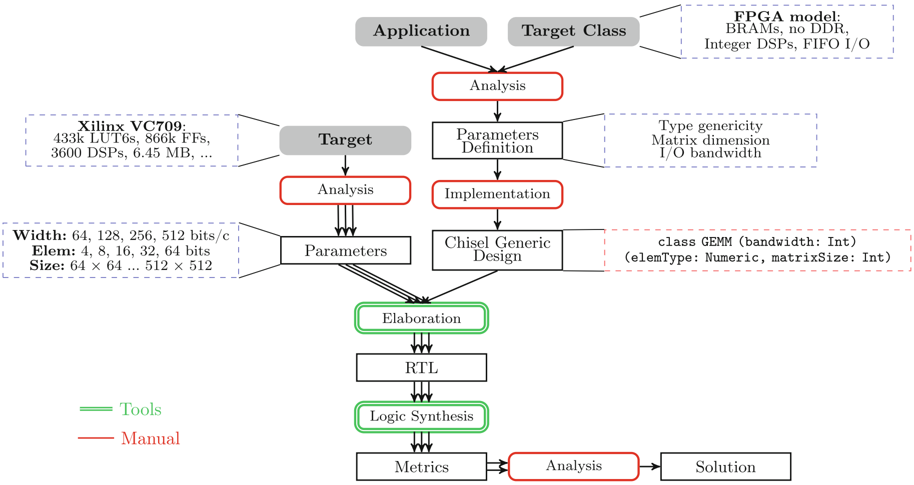


Fig. 1. Proposed methodology for design development. Dashed rectangles presents an example of application to the GEMM usecase.

Figure 1 presents this methodology. As can be observed, we consider three entry points when implementing an application to dedicated hardware: obviously, the **target** and the **application** itself, but also the **target class**, which is defined by generic characteristics we aim at using for this specific application (*e.g.* memory type and capacity, available operators, communication links, ...).

To use this methodology, one has to distinguish two main steps: implementing a generic design of the application using `Chisel`, and instantiating this design with parameter variations for a particular target. Doing this, the generic design - that uses some particular constructs, like embedded memory or DSP units - can be used to implement the application on various boards which includes those constructs. The methodology needs 3 main manual steps:

– Analysis of both application and target class, to define the parameters used for circuit parametrization
– Implementation of a generic architecture using `Chisel`
– Analysis of the target board, to define the parameter sets used for architecture instantiation, with respect to the target characteristics (*e.g.* the band width, the problem dimension, ...)

Elaboration and logic synthesis steps are done automatically, for each parameter set defined in this third step.

3 Methodology Usecase

In order to demonstrate both usability and advantages of the proposed methodology, we defined and implemented a generic GEMM architecture. It illustrates how preliminary reflexions on application and target class - communication model, available memory and computing units, temporal behavior, ... - can, with the help of `Chisel`, improve both productivity and code reusability with generic designs.

GEMM. The General Matrix Multiply (GEMM) algorithm is a generalization of the matrix product algorithm. Let A, B and C be square matrices of dimension $n \times n$ ($\mathcal{M}_n$), and $(\alpha, \beta) \in \mathbb{N}^2$. GEMM is defined as the following f function:

$$\begin{aligned} f\colon \mathbb{N} \times \mathbb{N} \times \mathcal{M}_n \times \mathcal{M}_n \times \mathcal{M}_n &\to \mathcal{M}_n \\ (\alpha, \beta, A, B, C) &\mapsto \alpha \cdot A \times B + \beta \cdot C \end{aligned} \tag{1}$$

Target Characteristics. For this implementation, we assume that the developer is targeting Xilinx FPGA technology. More precisely, this means that the developed design can take advantage of embedded operators for multiplication (*DSP block*) and **only** embedded memory (*Block RAMs*).

Application Study. GEMM computation complexity is $O(n^3)$ while its communication complexity is only $O(n^2)$. Since, in a generic context, matrices need to be sent to the design anyway, we can assume that $O(n^2)$ - *i.e.* the communication complexity - is a temporal complexity bound.

If one wants to reach this bound, it means that the implemented design needs to be able to compute matrix product in a temporal complexity of n^2. This defines the architecture parallelism level, as it requires to compute n scalar products ($\sum_{k=0}^{n} a_{i,k} \times b_{k,j}$ for $j \in [\![0; n-1]\!]$) in parallel to comply with it.

Figure 2 introduces the targeted temporal behavior for the implementation. It has been defined with respect to software considerations, as matrices are not interleaved nor transformed, except for B which has to be transposed for by-column access. Matrices are sent by blocks of size b (as defined in Sect. 3).

As one can observe on the Fig. 2, the input bus utilization is almost optimal (*i.e.* the input bus is almost used for the whole computation time), as results can be computed on-the-fly while A matrix is being streamed. This way, we can ensure that the induced design will be communication efficient.

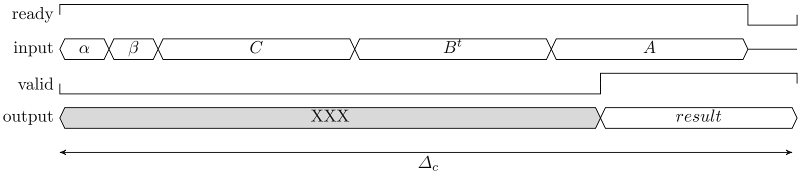


Fig. 2. Targeted chronogram for GEMM efficient implementation (Eq. 1)

Application-Specific Parameters. With such temporal behavior, we can compute the maximum throughput of a design implementing it, as a function of

- b the input (or output) bandwidth (in bits/cycle)
- f the clock frequency (in Hz)
- e the matrix element size (in bits)
- n the matrix dimension

We assume that a GEMM kernel performs $\rho = 2 \times n^3$ operations [10].

Let Δ_c be the number of cycles needed to compute the result of the GEMM algorithm - *i.e.* the delta cycle between sending α coefficient and receiving the last bit of the result matrix. We can state that $\Delta_c \geq 3 \times n^2 \times \frac{b}{e}$, $\frac{b}{e}$ being the number of elements sent per cycle, as three matrices must be sent.

Thus, the theoretical maximum throughput T, in number of operation per second (OPs), is given by

$$T = \frac{\rho \times f}{\Delta_c} \leq \frac{2 \times n^3 \times f}{3 \times n^2 \times \frac{b}{e}} = \frac{2}{3} \times \frac{f \times n \times e}{b} \qquad (2)$$

GEMM Parametrization. As specified in Sect. 2, the developer must think its architecture genericity before starting the implementation, to allow both exploration and design reutilization.

For GEMM application, we decided, after application and target class analysis, to define three parameters according to Sect. 3:

- b – bus bandwidth for input/output
- n – size of matrices
- type of elements (which bitwidth is defined as e).

4 Results

To study the usability and performances of our methodology, we implemented it as a tool flow, and used it to analyze and compare multiple GEMM architectures. This way, we can illustrate both hardware controlability and generation abilities of our methodology. We generate 80 different architectures, varying input bandwidth b (64, 128, 256 and 512 bits/cycle), element bitwidth e (4, 8, 16, 32 and 64 bits) and matrix dimension n (64×64, 128×128, 256×256 and 512×512), and we study the impact of those parameters on the performance and resource usage of generated designs. The architectures are generated, simulated and synthesized in less than 24 h, thanks to our tool flow (see Fig. 1).

4.1 Experimental Setup

For the experimentation, we targeted a **Xilinx VC709** board which includes Block RAMs and integer DSPs, as specified in Sect. 3. It embeds a `xc7vx690` FPGA with 433k LUT6s, 866 FFs, 3600 DSPs and 1470 BRAMs (6.45 MB).

We developed a tool flow implementing the methodology proposed in Sect. 2, based on `Chisel` (latest 3.2 version) as entry point to generate multiple Verilog designs with respect to the parameters defined in Sect. 3. The flow simulates generated designs behavior using `verilator` to ensure functionality, comparing it to a software reference defined in Scala. It also uses simulation to extract design latency $\varDelta_c$ as defined in the temporal behavior model (Fig. 2). After generation and simulation steps, we use Xilinx `vivado` (2017.3 version) to synthesize designs and extract performance and resource metrics. For performance evaluation, we use the estimated post-synthesis clock frequency and the simulation latency. We evaluate resource usage (LUT, Flip Flop, DSP and BRAM usage) thanks to post-synthesis resource report. All results presented in this section are given after `vivado` synthesis step.

Remark: We are only considering designs that can physically fit for this section, implying that Tables 2, 3 and Fig. 3a only include those designs. Figure 3b represents non-fittable designs as hatched.

4.2 Control of Generated Hardware

This section presents our designs achieved performance, demonstrating that this methodology allows to control generated accelerators composition.

GEMM Implementation. As stated in Sect. 1, GEMM algorithm can be implemented using various types and precision, SGEMM and DGEMM (using respectively IEEE-754 simple and double precision) are the most used version, as it has been widely used for performance comparison.

However, since we are targeting Xilinx FPGAs, which does not include dedicated floating point units, we chose to implement a fixed-point GEMM version here. Since the design generator includes type parameters, one could - with few changes to the control flow - target SGEMM and/or DGEMM variants once he implemented basic floating operations on Xilinx boards, as stated in Sect. 2.1.

Impact of Type Precision. Table 2 presents the influence of type precision on achieved throughput for GEMM implementation. For each element bitwidth e, we selected the most performant generated design, *i.e.* the design that offers the higher throughput, with the generation parameters (b, n) associated. We compared the throughput estimation (based on simulation) with the maximal theoretical throughput as defined in Eq. (2), indicating the functioning frequency of the generated designs in the last column, for information purpose.

By computing the theoretical differentials between achieved and theoretical throughput, we can note that generated designs achieve at least 92% of maximal throughput - for the 8 bits version - meaning that the input bus utilization is almost maximal, and that the behavior can be finely controlled from architecture design to generation.

We showed that our flow can be used to design, implement and analyze designs with high controlability on generated hardware, and demonstrated it on an analysis of type precision influence on the performances of GEMM implementations.

Table 2. Impact of element bitwidth on GEMM throughput.

Element (e)	I/O (b)	Size (n)	Throughput (GOps)		Frequency (MHz)
			Achieved	Theoretical (2)	
4	512	128	3680	3844	352.00
8	512	64	934	1016	372.72
16	128	512	700	701	256.74
32	128	256	226	227	331.34
64	128	256	66	68	197.78

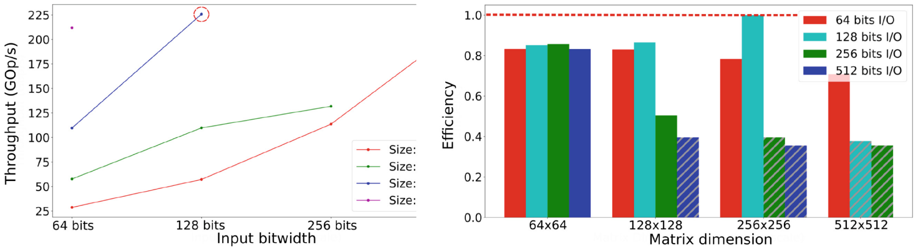
4.3 Architecture Exploration: Dimensioning the Application

We have shown that type precision has a considerable impact on the achievable throughput of generated designs. As type precision also impacts applicative performance metrics a developer can not always act on the type precision, that can be fixed by application specific needs.

In this section, we chose to target 32 bits fixed point GEMM implementation, though it is not comparable to SGEMM subroutine on 32 bits floating points because of the complexity of floating point operations (even with dedicated DSPs), to demonstrate the ability of our methodology to explore multiple parameter sets with no changes to the original `Chisel` description.

Figures 3a and b respectively compare generated designs throughput and efficiency for various sizes of input matrices, and various I/O bandwidths. Efficiency e is defined as $\frac{\rho}{f \times \Delta_c} \times \frac{1}{\|\%resource\|} = \frac{2 \times n^3}{f \times \Delta_c \times \|\%resource\|}$, $i.e.$ $\frac{performance}{resource}$ ratio[1], unified with respect to computational complexity specified in Sect. 3. Using a n^3 factor allows to compare solution of different dimensions, since computing GEMM in $\mathcal{M}_n$ space is equivalent to 8 computations in $\mathcal{M}_{\frac{n}{2}}$. We can observe that for 32 bits implementations, the most efficient version (Fig. 3b) is to operate on matrix kernels of size 256×256, using 2×128 bits/cycle I/O, with an achieved throughput of 224 GOP/s - among the 80 architectures generated. For this design, clock frequency reach 331.4 MHz, the theoretical optimal bit rate is thus 2.8 GB/s. On the other hand, Fig. 3a shows that the most performant design is to operate on kernels of size 128×128, with 2×128 bits/cycle I/O, achieving a throughput of 226 GOP/s.

By generating and comparing those GEMM architectures, we showed that our generation flow allows to determine which architecture is the most performant or the most efficient, with respect to constraints one might have on integration.



(a) Performance comparison (b) Standardized efficiency comparison

Fig. 3. Metric comparison on 32 bits GEMM versions

4.4 Existing Solutions

GEMM implementations have been widely used to compare platform performances, as well as implementation choices. We propose to compare our implementations to GEMM instances on various platforms.

[1] Resource metric is defined as the maximum usage percentage for the 4 considered resources: *LUTs*, *Flip Flops*, *BRAMs* and *DSPs*.

fBLAS [4] implements both SGEMM and DGEMM variants, using HLS on two Intel Altera FPGAs. It is important to note that Intel Altera FPGAs embed dedicated floating point DSP, while Xilinx FPGAs does not include any floating point dedicated units.

Garg et al. [6] propose hybrid CPU-GPU SGEMM and DGEMM implementations based on Intel Ivy Bridge and AMD Richland platforms.

Our custom solutions present the most performant designs we generated for precision on 16, 32 and 64 bits. As VC709 target does not include floating point units, SGEMM and DGEMM will be compared respectively to fixed point solutions on 32 and 64 bits. We also choose to study results on 16 bits fixed point, since applicative accuracy needs might be compatible with lower precision type.

For each solution in Table 3, we can observe different implementations - variation of platform, target and type precision - and the associated achieved performances, given as implementation throughput. We can see that both our 32 and 64 bits custom versions are comparable in term of performance with the hybrid solution of [6], and with the fBLAS solution on Intel Altera Arria 10, if a fixed point solution is acceptable for a given application needs. Stratix 10 board being way wider than a VC709, we can not compare solutions fairly.

Table 3. Throughput for GEMM implementations on different platforms

Solution	Platform	Target	Precision	Throughput
Custom[a]	FPGA	VC709	16 bits	700 GOps
			32 bits	226 GOps
			64 bits	68 GOps
fBLAS [4]	FPGA	Arria 10	32 bits	200 GFlops
			64 bits	25 GFlops
		Stratix 10	32 bits	750 GFlops
			64 bits	75 GFlops
Hybrid [6]	CPU-only	Ivy Bridge	32 bits	170 GFlops
		Richland	32 bits	80 GFlops
			64 bits	40 GFlops
	GPU-only	Ivy Bridge	32 bits	140 GFlops
		Richland	32 bits	274 GFlops
			64 bits	27.3 GFlops
	CPU + GPU	Ivy Bridge	32 bits	235 GFlops
		Richland	32 bits	274 GFlops
			64 bits	57.4 GFlops

[a] Fixed point precision is used instead of floating point, as stated earlier.

4.5 Analysis and Contribution

With those experiments, we show that with a tool flow based on `Chisel` and a particular methodology, we are able to easily define a generic GEMM accelerator kernel, which presents multiple advantages when compared to HLS generated or RTL written ones.

We demonstrate that using `Chisel` allows a high controlability on generated hardware, and that, with a sufficient knowledge on hardware development, one can easily describe a precise architecture, with no worry on which inferences the tool flow could make when generating the circuit.

We also show that using generic architectures can be useful when it comes to evaluation of parameters influences, and that using higher level of abstractions, hardware developers can easily compare architectural trade-offs for a given application, in order to take the best of the available resources.

5 Conclusion

In this paper, we introduce a design methodology associated to a toolflow that can be used to implement computation kernels on FPGAs with higher abstraction level.

We demonstrate the functionality of this new tool through a use case on GEMM algorithm, which is highly representative for all algebra computations. We show that we can define generic architecture descriptions with parametrization, thanks to `Chisel`, allowing architecture generation and comparison.

Generated GEMM implementations performances are comparable to designs generated with HLS methodology, as well as CPU and/or GPU solutions proposed in the literature.

We now aim at reusing developed GEMM kernels to implement efficient, configurable and highly generic CNNs using the presented framework.
We also want to implement other computation kernels to study influence of applications and target environments on resource usage and achieved performances.

References

1. Alon, E., Asanović, K., Bachrach, J., Nikolić, B.: Invited: Open-Source EDA Tools and IP, A View from the Trenches, p. 3 (2019)
2. Bachrach, J., et al.: Chisel: constructing hardware in a Scala embedded language. In: Proceedings of the 49th Annual Design Automation Conference on - DAC 2012, San Francisco, California, p. 1216. ACM Press (2012)
3. Caulfield, A.M., et al.: A Cloud-Scale Acceleration Architecture, p. 13 (2016)
4. De Matteis, T., de Fine Licht, J., Hoefler, T.: FBLAS: streaming linear algebra on FPGA. arXiv:1907.07929 [cs], August 2019
5. Dongarra, J.J., Du Croz, J., Hammarling, S., Duff, I.S.: A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Softw. **16**(1), 1–17 (1990)

6. Garg, R., Hendren, L.: A portable and high-performance general matrix-multiply (GEMM) library for GPUs and single-chip CPU/GPU systems. In: 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Torino, Italy, pp. 672–680. IEEE, February 2014
7. Ouyang, J., Lin, S., Qi, W., Wang, Y., Yu, B., Jiang, S.: SDA: software-defined accelerator for large-scale DNN systems. In: 2014 IEEE Hot Chips 26 Symposium (HCS), Cupertino, CA, USA, pp. 1–23. IEEE, August 2014
8. Koenig, J., Biancolin, D., Bachrach, J., Asanovic, K.: A hardware accelerator for computing an exact dot product. In: 2017 IEEE 24th Symposium on Computer Arithmetic (ARITH), London, United Kingdom, pp. 114–121. IEEE, July 2017
9. Pedram, A., Gerstlauer, A., van de Geijn, R.A.: A high-performance, low-power linear algebra core. In: ASAP 2011–22nd IEEE International Conference on Application-Specific Systems, Architectures and Processors, Santa Monica, CA, USA, pp. 35–42. IEEE, September 2011
10. Underwood, K.D., Hemmert, K.S.: Chapter 31 - The implications of floating point for FPGAs. In: Hauck, S., Dehon, A. (eds.) Reconfigurable Computing, pp. 671–695. Systems on Silicon, Morgan Kaufmann, Burlington (2008)
11. Zhao, Z., Hoe, J.C.: Using Vivado-HLS for structural design: a NoC case study. arXiv:1710.10290 [cs], October 2017