POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY

ON HOW TO EFFECTIVELY TARGET FPGAS FROM DOMAIN SPECIFIC TOOLS

Doctoral Dissertation of:
Emanuele Del Sozzo

Supervisor:
Prof. Marco D. Santambrogio

Tutor:
Prof. Cristiana Bolchini

The Chair of the Doctoral Program:
Prof. Andrea Bonarini

2018 – XXXI Cycle

To my Family

Abstract

HETEROGENEOUS System Architectures (HSAs) represent a promising solution to face the limitations of modern homogenous architectures, in terms of both performance and power efficiency. Indeed, thanks to the combination of hardware accelerators like GPUs, FPGAs, and dedicated ASICs, such systems are able to efficiently run performance demanding applications belonging to different application scenarios (like image and signal processing, linear algebra, computational biology, etc.) on the most suitable device for that domain. In order to fully take advantage of HSAs, in the last years new programming models and tools able to efficiently target such architectures, in terms of both final performance and productivity, emerged. Domain Specific Languages (DSLs) and Machine Learning (ML) frameworks are two significant examples. Both permit users to quickly and easily develop portable and efficient designs for multiple architectures. However, although DSLs and ML frameworks are highly effective in assisting users towards the generation of efficient designs for CPUs and GPUs, they still lack a concrete support for FPGAs. Indeed, even though FPGA toolchains have significantly improved and increased their features over the last years, the whole FPGA design process remains complex and the integration with high-productivity tools and languages is still limited. For these reasons, this research project focuses on the development of tools able to efficiently and easily target FPGAs from domain-specific scenarios. In particular, it consists in both a framework for the fast-prototyping and deployment of CNN accelerators on FPGA, and FROST, a unified backend to efficiently hardware-accelerate DSLs on FP-

GAs. On one hand, the goal of the CNN framework is to bridge the gap between high-productivity ML frameworks, like TensorFlow and Caffe, and FPGA design process. The framework automatizes the CNN implementation flow on FPGA, supports Caffe descriptions of the network, and provides a C++ library to design dataflow accelerators, as well as an integration with TensorFlow to train the network. On the other, starting from an algorithm described in one of the supported DSLs, FROST translates it into its Intermediate Representation (IR), performs a series of FPGA-oriented optimizations steps, and, finally, generates an optimized design suitable of FPGA tools. In order to better leverage the features of the FPGA and enhance the performance, FROST provides a high-level scheduling co-language the user can exploit to guide the optimizations to apply, as well as specify the architecture to implement. This allows to easily evaluate different hardware designs and choose the most suitable to the input algorithm.

Sommario

I Sistemi di Architetture Eterogenee (HSA) rappresentano una soluzione promettente per fronteggiare le limitazioni delle moderne architetture omogenee, in termini sia di prestazioni che di efficienza dal punto di vista della potenza. Infatti, grazie alla combinazione di acceleratori hardware come GPU, FPGA e ASIC dedicati, tali sistemi sono in grado di eseguire efficientemente applicazioni che richiedono alte prestazioni e che appartengono a diversi scenari applicativi (come il processamento di immagini e segnali, l’algebra lineare, la biologia computazionale, e così via) sul dispositivo più adatto per quel dominio. Al fine di sfruttare a pieno gli HSA, negli ultimi anni sono emersi nuovi modelli e strumenti di programmazione in grado di rivolgersi a tali architetture, in termini sia di prestazioni finali che di produttività. Linguaggi a Dominio Specifico (DSL) e framework di Machine Learning (ML) sono due esempi significativi. Entrambi permettono all’utente di sviluppare velocemente e facilmente design portabili ed efficienti per architetture multiple. Tuttavia, nonostante DSL e framework di ML sono altamente efficienti per CPU e GPU, non lo sono altrettanto per FPGA. Infatti, anche se, nel corso degli ultimi anni, le toolchain per FPGA sono significantivamente migliorate ed hanno aumentato le loro caratteristiche, l’intero processo di design per FPGA resta complesso e l’integrazione con strumenti e linguaggi ad alta produttività è ancora limitato. Per queste ragioni, questo progetto di ricerca si concentra sullo sviluppo di strumenti in grado di rivolgersi in maniera efficiente e facile alle FPGA partendo da scenario a dominio specifico. In particolare, questo progetto consiste in sia un framework per lo sviluppo veloce di acceleratori per CNN su FP-

GA, e FROST, un backend unificato per accelerare efficientemente i DSL su FPGA. Da un lato, lo scopo del framework per le CNN è di colmare lo spazio tra i framework di ML ad alta produttività, come TensorFlow e Caffe, e il processo di design per FPGA. Il framework automatizza il flusso di implementazione di CNN su FPGA, supporta descrizioni in Caffe della rete, e fornisce una libreria C++ per sviluppare acceleratori dataflow, insieme ad una integrazione con TensorFlow per allenare la rete. Dall'altro lato, partendo da un algoritmo descritto in one dei DSL supportati, FROST lo traduce nella propria rappresentazione intermedia (IR), applica una serie di passi di ottimizzazioni orientati alle FPGA, e, infine, genera una implementazione ottimizzata adatta per gli strumenti per FPGA. Al fine di sfruttare al meglio le caratteristiche della FPGA e migliorare le prestazioni, FROST fornisce un co-linguaggio di scheduling ad alto livello che l'utente può sfruttare per guidare le ottimizzazioni da applicare, e specificare l'architettura da implementare. Questo permette di valutare facilmente design hardware differenti e scegliere la più adatta l'algoritmo in input.

Contents

CHAPTER 1

Introduction

Thanks to the improvements in technology and methodologies, nowadays computing performance has a critical role in many and different fields, like finance, medical science, cutting-edge research, and so on. In the past decades, the main method used to improve the performance of computing systems consisted in increasing the operating frequency of the processing units. This allowed the users to benefit of such performance boost in a completely transparent way, with no change required to their applications. In this context, the improvements in transistor technologies were the fundamental factor that enabled the performance enhancement in computing systems. Indeed, according to Moore's law [1] and Dennard's scaling law [2], the yearly reduction of MOS transistors size permitted to fit more transistors within the same area of an integrated circuit, keeping the power profile roughly constant. Figure 1.1 portraits the trend of transistor count in Intel microprocessors from 1971 to 2016 [3]. This chart shows that, in about 45 years, the number of transistors inside Intel's processors increased from 2300 (Intel 4004 Processor [4], 1971) to 5.56 billion (Intel Xeon Phi Processor 7290F [5], 2016). However, the aforementioned trend is not valid anymore; indeed, due to the density of transistors, as well as the high operating frequencies, it is unfeasible for the processing units to dissipate the

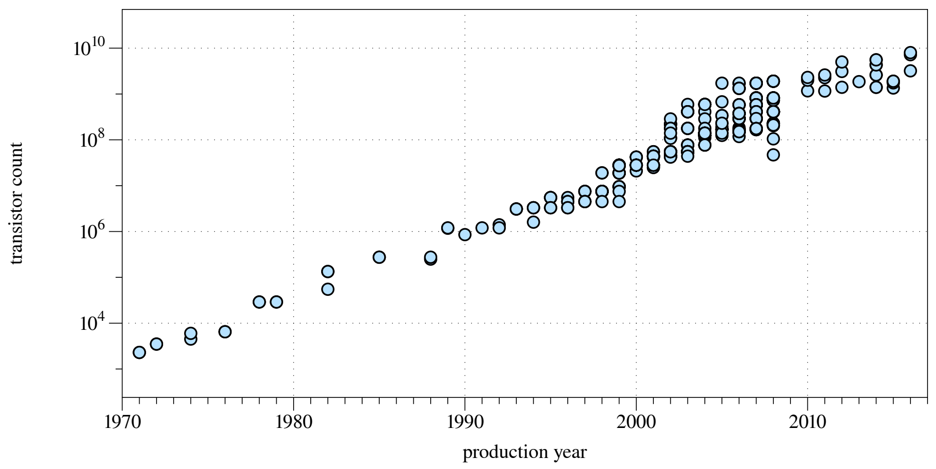

Figure 1.1: *Transistor count in Intel microprocessors from 1971 to 2016.*

significant thermal power produced [6, 7]. For this reason, academia and companies started to consider alternative approaches to face the power wall limit.

One of the proposed solutions oriented not only to power efficiency but also to performance demand consisted in adopting multi-cores and parallel systems [8–11]. This kind of architecture is currently widespread and available in different versions according to costumers’ needs. Indeed, it is straightforward to find, on one hand, quad-core processors on consumer electronics (for instance, Intel i5 [12] and i7 [13] series or AMD A-Series [14]) and, on the other, 16-cores on enterprise class server nodes (like Intel Xeon series [15], IBM Power Systems [16], Oracle SPARC Systems [17], and so on). However, differently from the frequency scaling, the performance improvement is up to the programmers, that have to reshape and adapt their applications to efficiently exploit multi-core systems. In this case, the law stated by Gene Amdahl in 1967 [18], which determines the diminishing returns to increasing the number of processors, is still valid today and limits the speedup provided by multi-core systems. As matter of fact, according to Amdahl’s law, the sequential part of the application bounds the theoretical speedup from parallelism. In other words, if 1/4 of the application is serial, the programmer can parallelize only 3/4 of the application, achieving a maximum speedup of 4, even using hundreds of processors.

Figure 1.2 displays the impact of Moore’s law, Dennard scaling and Amdahl’s law on processor performance for the past 45 years. In this scenario,

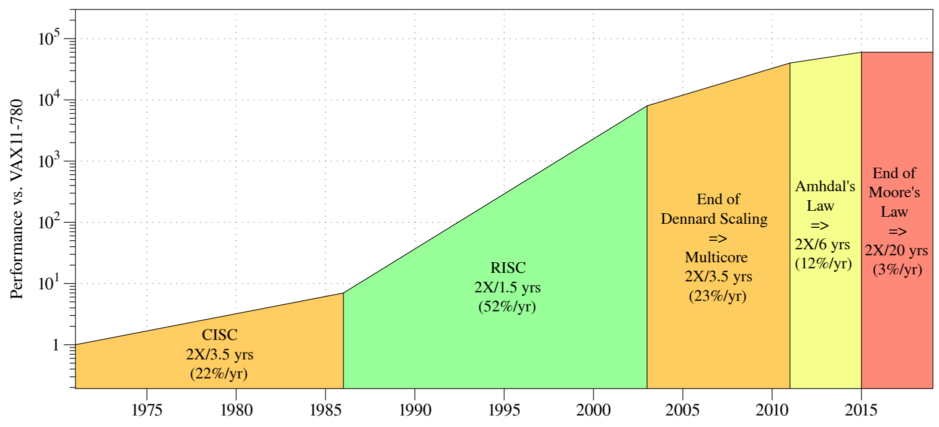

Figure 1.2: *Average performance gain for a single program over time versus VAX 11-780 using SPECintCPU [21]. Image from [22].*

if this trend does not change, it would take 20 years for a single-program performance to double. In summary, at the present state of the art [19]:

- transistors improvement is getting significantly slow (due to the end of Moore's Law),
- the peak power per mm^2 of chip is growing (due to the end of Dennard scaling), but factors like electromigration, mechanical and thermal bounds limit the power budget per chip, and
- we have already attempted the multi-core solution (limited by Amdahl's Law).

Considering these inevitable limitation, a promising approach to pursue performance while keeping energy consumption under control is the exploitation of Heterogeneous System Architectures (HSAs) [20].

1.1 Heterogeneous Computing

Heterogeneous architectures represent a promising solution to face the limitations of modern homogeneous architectures, in terms of both performance and power efficiency. Indeed, thanks to the combination of hardware accelerators like Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), and dedicated Application Specific Integrated Circuits (ASICs), such systems are able to efficiently run performance demanding applications belonging to different application scenarios (like image and signal processing, linear algebra, computational biology, etc.) on the most suitable device for that domain. CPUs can efficiently run generic tasks, GPUs are optimal for massively parallel repetitive tasks, and FPGAs can be (dynamically) configured to provide a hardware implementation of a software description for an efficient execution. As a result, thanks to their nature, HSAs may be used to satisfy different workload goals, in terms of both performance and power/energy consumption. For these reasons, HSAs are currently in use also in High Performance Computings (HPCs) systems; indeed, the top supercomputer in July 2018, according TOP500 list [23], is *Summit*, a system that features, per each node, both an IBM Power9 processor and an NVIDIA Volta GPU [24], while other supercomputers, like *Tianhe-2A*, are accelerated by a Matrix-2000 coprocessor [25]. On the other hand, HSAs appear also in July 2018 Green500 list [26], the ranking of the most energy-efficient supercomputers in the world. HSAs dominate the top places of the Green 500 list; indeed, the most energy-efficient supercomputer is *Shoubu system B*, a heterogeneous system composed of Intel Xeon D-1571 CPU [27] and PEZY-SC2 [28] many-core accelerators, which also feature second and third ranked supercomputers in this list.

Naturally, if hardware evolves, software has to evolve as well. Thus, in order to fully take advantage of HSAs, we need new programming models and tools able to efficiently target such architectures, in terms of both final performance and productivity. For this reason, in the last years we have witnessed the rise of many and different solutions designed to simplify the development of applications for multiple target architectures [29–31]. In this context, Domain Specific Languages (DSLs) represent one of the most interesting solutions [32, 33]. Indeed, current DSLs allow the user to quickly and easily develop portable designs for multiple architectures. Thanks to the restriction of the domain, DSL compilers are able to rapidly explore the design space and deeply optimize the resulting implementations. As a result, DSL applications often outperform hand-tuned libraries. On the other hand, Machine Learning (ML) frameworks represent another fascinating solution. Even though it has been around for decades, ML has been one of the major topics in research and engineering field over the last years [34]. The reason for that is, on one hand, the availability of a huge amount of data to train ML algorithms, and, on the other, the possibility to efficient execute ML algorithms, like Convolutional Neural Networks (CNNs), on hardware. As a consequence, ML tools quickly evolved. Nowadays, frameworks like TensorFlow [35], Caffe [36], and Torch [37] offer efficient solutions to easily both implement ML algorithms, without a significant expertise of the field, and target hardware accelerators.

1.2 Problem Statement

Although DSLs and ML frameworks are highly effective in assisting users towards the generation of efficient designs for CPUs and GPUs, they still lack a concrete support for FPGAs. Historically, hardware design for FPGAs has always been more complex with respect to the design for CPUs and GPUs. This limited the adoption of FPGAs in datacenters and HPC systems, in spite of the great design opportunities FPGAs can provide in such contexts (like arbitrary data precision and the possibility to create a custom architecture, basically a Domain Specific Architecture, tailored to the target application scenario). Similarly to what happened for CPUs and GPUs, over the last years FPGA toolchains have significantly improved and increased their features. For instance, High-Level Synthesis (HLS) tools facilitated the design on FPGA; indeed, they permit to hardware accelerate algorithms using languages like C/C++ and OpenCL, instead of Hardware Description Languages (HDLs) like Verilog and VHDL. However, the whole FPGA design process remains complex and the integration with high-productivity tools and languages is still limited. In particular, on one hand, even though there exist some DSLs able to target FPGA [38–42], a common solution capable of supporting multiple DSLs, even the ones that do not have an FPGA backend, is still lacking. On the other, an official and fully-integrated FPGA support within industrial ML frameworks, like TensorFlow, is not available yet.

1.3 Contributions

Given these motivations, this thesis focuses on the development of tools able to efficiently and easily target FPGAs from domain specific scenarios. In other words, the goal of this thesis is to demonstrate the efficiency of FPGAs when applied in a well-defined context, where the user can transparently take advantage of such devices and the tools manage all the complexity (i.e. the generation of an efficient hardware design). In particular, this work describes tools oriented to the hardware acceleration on FPGA of CNNs and DSLs. The purpose of such tools is to simplify the design of FPGA accelerators providing users with high-level abstractions to define the computation (and not its implementation). On the other hand, the proposed tools are able to build efficient hardware designs thanks to the restricted domain.

The first tool is a framework for the fast-prototyping and deployment of CNN accelerators on FPGA [43,44]. The goal of the framework is to bridge the gap between high-productivity ML frameworks, like TensorFlow and Caffe, and FPGA design process. The main features of the framework are:

- A novel framework written in Python, providing a set of modules that implement the toolchain for the design and the implementation of CNNs on FPGAs;
- A flexible internal representation based on Google Protocol Buffers that is compliant with a subset of the layer definitions of the Caffe deep learning framework, giving the possibility to provide existing models as input;
- The integration with TensorFlow for CNN training, providing the training set and the test set directly to the framework;
- A hardware library with customizable modules implementing the different type of layers of CNNs.

The second tool is *FROST*, a unified backend to efficiently hardware-accelerate DSLs on FPGAs [45, 46]. Starting from an algorithm described in one of the supported DSLs, FROST translates it into its Intermediate Representation (IR), performs a series of FPGA-oriented optimizations steps, and, finally, generates an optimized design for HLS tools. Here the main features of FROST:

- A common backend exposing an IR that DSLs can target in order to accelerate their computations on FPGA;
- Support for Halide [32], a state-of-the-art DSL for image processing, and Tiramisu [47], a code optimization framework for HPC systems;
- A high-level scheduling co-language the user can exploit to guide the optimizations to apply, specify the architecture to implement, and combine with the optimizations offered by the frontends;
- Generation of efficient hardware designs suitable for HLS tools.

1.3.1 Publications

These paper partially contain the contributions of this thesis:

- **E. Del Sozzo**, A. Solazzo, A. Miele, and M. D. Santambrogio, *"On the Automation of High-Level Synthesis of Convolutional Neural Networks"*, 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), May 2016

- A. Solazzo, **E. Del Sozzo**, I. De Rose, M. De Silvestri, G. C. Durelli, and M. D. Santambrogio, *“Hardware Design Automation of Convolutional Neural Networks"*, 2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), July 2016
- **E. Del Sozzo**, R. Baghdadi, S. Amarasinghe and M. D. Santambrogio, *“A Common Backend for Hardware Acceleration on FPGA"*, 2017 IEEE International Conference on Computer Design (ICCD), November 2017
- **E. Del Sozzo**, R. Baghdadi, S. Amarasinghe and M. D. Santambrogio, *“A Unified Backend for Targeting FPGAs from DSLs"*, 2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP), July 2018
- R. Baghdadi, J. Ray, M. B. Romdhane, **E. Del Sozzo**, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, S. P. Amarasinghe, *“TIRAMISU: A Polyhedral Compiler for Expressing Fast and Portable Code"*, 2019 ACM International Symposium on Code Generation and Optimization (CGO), February 2019

1.4 Thesis Organization

The work presented in our thesis is organized as follows:

- Chapter 2 reports the architecture and features of GPUs and FPGAs and compares different solutions for HSAs according to multiple metrics in order to identify the best trade-off;
- Chapter 3 presents the state-of-the-art languages and tools to develop hardware designs for FPGAs, and stresses the motivations of this work;
- Chapter 4 describes a framework to automate the hardware acceleration on FPGA of Convolutional Neural Networks;
- Chapter 5 focuses on *FROST*, a common backend for targeting FPGAs from Domain Specific Languages;
- Chapter 6 gives a general overview of this thesis work, and provides some insights on the possible future work.

CHAPTER 2

Background on Hardware Architectures

The complexity and performance demand of current computing systems, as well as their power consumption, are significantly growing at a remarkable rate, and the advancements in silicon technologies and manufacture process cannot any longer the yearly multiplying of computing performance typical of the previous decades. In fact, the innovations in transistor technologies, according to Moore's law [1] and Dennard's scaling law [2], allowed such a fast improvement in microprocessor performance, enabling more transistors to fit in the same area of the integrated circuit, maintaining a generally constant power density. Along with the advancement of transistor technology, a significant effort in the progress of micro-architectural techniques permitted to leverage an ever growing amount of instruction level parallelism, while the development of on-die cache memories allowed a fast access to frequently accessed data, hiding the latency between the fast microprocessor and the slow DRAM memory. However, as we are approaching the end of Moore's law and Dennard's scaling law, this trend is not true anymore; indeed, it is not possible to further lower the threshold voltage of transistor while keeping the leakage current under control. Moreover, the ever decreasing physical size of transistors generates additional scaling challenges; indeed, as the power densities of microprocessors is reach-

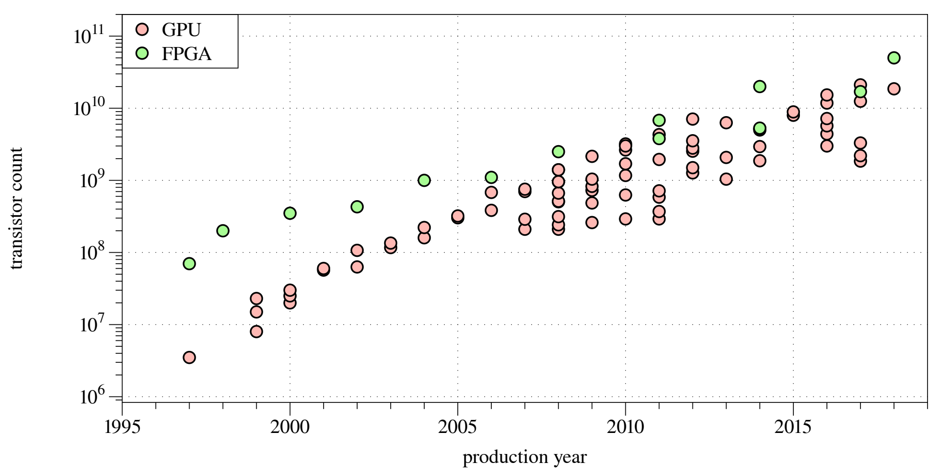

Figure 2.1: *Transistor count in GPUs and FPGAs devices.*

ing those of nuclear reactors, the power consumption has become a hard limit the microprocessors evolution [6]. For these reasons, since the single-thread performance stabilized, research shifted the architectural paradigm to keep on improve system performance. This paradigm shift consisted in developing multi- and many-core architectures, as well as leveraging thread-level parallelism instead of instruction-level parallelism. As a result, multi-core systems featuring a set of simple cores were able not only to outperform highly complex single-core superscalar processor, but also to respect the same power budget. The large amount of applications containing parallelizable task represents one of the main reasons of multi-core systems widespread success. In spite of these advantages, the dark silicon phenomenon is constraining the growth of multi-core architectures [48,49]. As matter of fact, also in this case the power budget limits the number of cores a chip can contain. This leads to a under-utilization of transistors, preventing multi-core systems to further scale and, consequently, creating new design challenges.

In this scenario, Heterogeneous System Architectures (HSAs) represents an emerging approach to offer high-performance solutions for both consumers and High Performance Computing (HPC) systems. According to trends and projections [50], heterogeneous architectures seem to be the most feasible way to reach exascale performance in HPC systems, while maintaining a manageable power budget. HSAs are to improve the overall performance of a system thanks to the combination of different kinds of processing units, each one in charge of execution the most suitable task.

In general, a typical HSA combines a general purpose multi-core Central Processing Unit (CPU), which executes the control-intensive parts of the application, with highly efficient hardware accelerators whose purpose is to run the most computationally intensive and performance-oriented tasks of the application. The main hardware devices employed as accelerators are, usually, Application Specific Integrated Circuits (ASICs), Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs). Just like CPUs, these devices evolved through the years, from both a hardware (Figure 2.1) and a software prospective. In this Chapter, we first describe the architecture of GPUs and FPGAs, then we analyze and compare the advantages and disadvantages of both solutions against CPUs and ASICs, in order to identify which architecture offers the best trade-off according to different metrics.

2.1 Graphics Processing Unit

Graphics Processing Units (GPUs) are parallel devices designed to execute both traditional graphics computations and general-purpose tasks. Even though their main computational domain was computer graphics, in the last years GPUs became more and more popular in equally demanding fields such as biology, finance and engineering [51]. Nowadays, GPUs are the most popular choice when it comes to accelerating such domains. The fundamental event in this shift towards general-purpose GPU was the disclosure of the NVIDIA's Compute Unified Device Architecture (CUDA) platform in 2006 [30]. CUDA is a parallel platform and programming model that enables remarkable increases in performance by leveraging the computational power of the GPUs. This has led to the development of other multiplatform frameworks, like OpenCL [29], able to support also GPUs from other vendors, such as AMD [52]. The outcome is a robust and efficient support to the most widely used libraries in fields like computational sciences, data analysis and linear algebra, as well as a stable integration within some of the most used programming languages, like C, C++, Fortran, Python and MATLAB. GPU architecture contains different components such as a unified shader pipeline, an Arithmetic Logic Units (ALUs) built in compliance with the IEEE standard for floating-point arithmetic, an instruction set designed for general computation, and execution units with arbitrary read and write access to standard and shared memory. Modern GPUs includes such features in order to excel at general-purpose computation. Thanks to their architecture, GPUs permit to leverage parallelism and reach a high level of performance. Figure 2.2 displays the difference

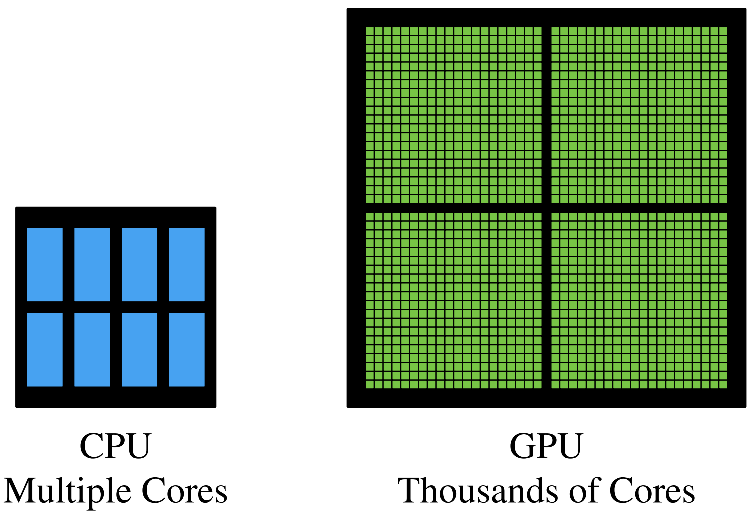

Figure 2.2: *Comparison between CPU and GPU architectures.*

between the architectures of CPU and a GPU in terms of number of cores. More specifically, on one hand, a CPU consists of a few cores designed for complex and sequential serial processing, on the other, the massively parallel architecture of a GPU contains thousands of smaller, more efficient cores optimized for managing multiple tasks simultaneously.

Although the GPU architecture permits to process a huge amount of data in parallel, it has some limitations. For instance, while GPUs can easily outperform CPUs with data-level parallelism applications, their simple architecture is slower when executing pure sequential and control-intensive algorithms. For this reason, in order to take advantage of GPUs, it is often necessary to significantly reshape CPUs well-designed algorithms, and small changes in code can generate relevant orders of performance due to architectural constraints. Besides, a GPU cannot automatically store data on disk when the memory is full, and it operates on vectors of integers and floats, but it cannot work on strings, characters or other data structures. Moreover, GPUs result less efficient than CPUs when dealing with control-intensive tasks. Finally, since the data transfer between GPUs and CPUs could be a bottleneck, GPUs have to process high amounts of parallel tasks, to leverage the higher amount of (slower) processing units and to minimize the data transfer latency.

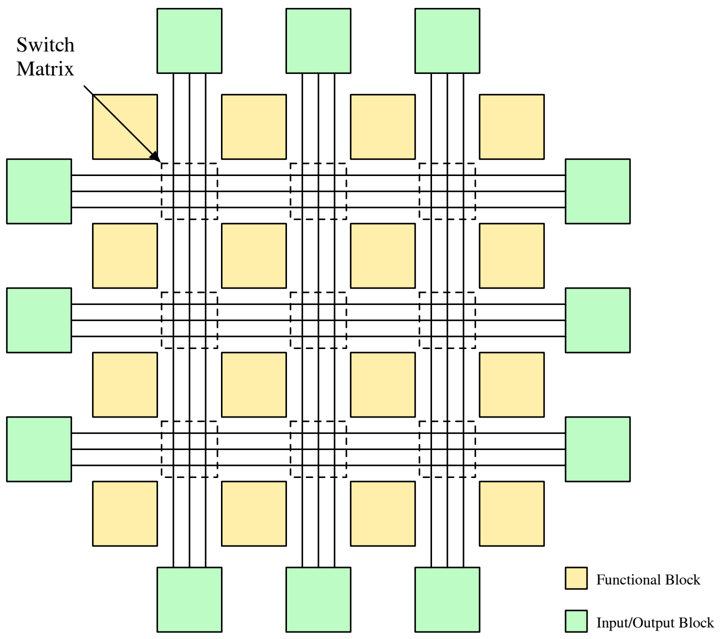

Figure 2.3: *High-level structure of an FPGA.*

2.2 Field Programmable Gate Array

A Field Programmable Gate Array is an integrated circuit whose main feature is consist in being electrically programmable after fabrication. This permits to change the behavior of the FPGA and implement nearly any type of digital circuit [53]. Such a feature makes the FPGA an ideal device for many and different applications, ranging from testing and prototyping ASIC, to the production of cheap hardware circuits. Another field of application of FPGAs is *reconfigurable computing*. In this context, the users exploit the FPGA as a hardware accelerator and take advantage of its flexibility to program the device with different applications at different moment, achieving high performance typical of hardware implementations.

Even though there is not a standard terminology, as it changes among vendors, from a high-level prospective an FPGA device contains:

- A bi-dimensional matrix of functional blocks to implement logic functions as well as provide on-chip memory;
- A programmable interconnection network to connect the functional blocks;

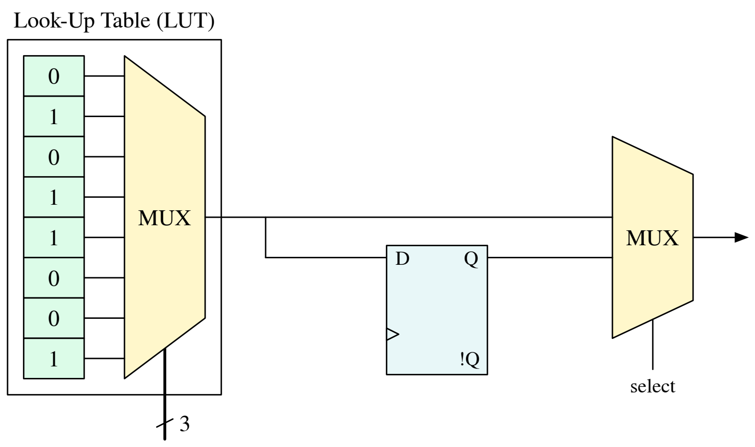

Figure 2.4: *Simplified structure of a CLB with a* 3*-input LUT.*

- Input/Output Blocks (IOBs) at the periphery of the FPGA, which operate as interfaces between the device and the external world.

Figure 2.3 portraits the architecture of an FPGAs. In the next Sections we describe the main components available on an FPGA device, as well as its configuration and design process.

2.2.1 Configurable Logic Blocks

The basic functional block of an FPGA is the Configurable Logic Block (CLB). This element both implements combinational or sequential functions, and provides a basic storage capability. The main building block of a CLB is a configurable combinational circuit, usually implemented by means of a Look-Up Table (LUT), a device capable of storing any n-input combinational function. A basic LUT is a 2^n-bit memory element incorporating the truth table of the combinational function, along with a multiplexer to select the proper output bit from the truth table according to the n-bit input. The combination of multiple LUTs permits to build more complicated functions. Modern FPGAs organizes LUTs within CLB slices, which communicates with the other processing blocks by means of the interconnection network. An important design trade-off consists in the size of the LUTs; indeed, a proper LUT size may balance area and delay. Typically, the size and the number of the LUTs within the CLBs change according to the target market of the FPGA, as well as the vendor. For instance, each CLB within Xilinx 7 Series FPGAs contains two slices, each one comprises four 6-input 2-output LUTs [54]. In order to implement sequential circuits, CLBs also

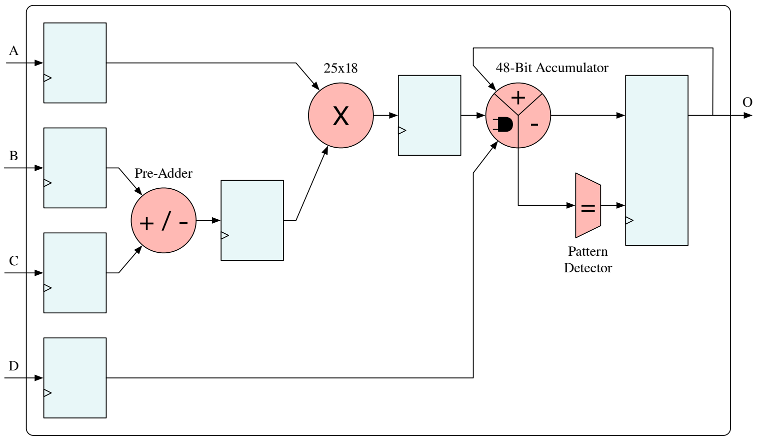

Figure 2.5: *The structure of a Xilinx DSP48E1 Slice.*

include memory elements. Thus, a basic CLB usually contains, in addition to the LUT, a D-type Flip Flop (FF) and a multiplexer to select the proper output between the FF and the LUT.

2.2.2 Digital Signal Processors

While in the past FPGAs were homogeneous and contained only general-purpose CLBs as functional blocks, modern FPGAs are heterogeneous and, alongside CLBs, feature specialized elements to efficiently execute some kinds of computation. As matter of fact, although modern FPGAs feature a impressive number of CLBs, they usually result space and time inefficient when implementing computationally intensive operations, in particular floating-point ones. In this scenario, Digital Signal Processors (DSPs) represent a commonly available example of specialized functional blocks. Figure 2.5 display the architecture of a Xilinx DSP48E1 slice. The purpose of DSPs is to mathematically process and manipulate input signals, in order to perform such operations more efficiently than CLBs. DSPs are capable of implementing several arithmetic operations, ranging from additions/subtractions to multiply and accumulate operations. This allows FPGAs to implement fully custom, pipelined designs for computationally intensive applications, reaching both maximum performance and high frequencies.

2.2.3 Block RAMs

FPGAs contain IOBs and pins in charge of connecting the device with off-chip peripherals. In case of applications demanding a significant data re-usage, the connection with components like DDR and Flash memories can satisfy the memory requirements. Nonetheless, the limited bandwidth of off-chip memories can constraint the application performance, as well as introduce stalls on the computational data-path. To overcome this limitation, in addition to CLBs and DSPs, modern FPGAs also feature a large amount of on-chip SRAM, called Block RAM (BRAM). The usages of such memories include single or dual port RAM, First-In First-Out (FIFO) functions, Finite State Machines (FSMs), and so on. The combination of multiple BRAMs connected together permits to implement memories capable of storing a relatively large amount of data. As matter of fact, the BRAMs of modern FPGAs can store several Megabytes.

2.2.4 Interconnection Network

The FPGA interconnection network consists of wires and programmable switches, and covers up to 90% of the whole area. A highly flexible network is a fundamental part of FPGA device, as it permits to connect all the aforementioned functional blocks available on the device and implement many classes of circuits with varying interconnection requirements. To this end, the interconnection network has to be definitely efficient, in order to interconnect both local and distant blocks while satisfying timing constraints of the application. Moreover, a modern FPGA with various functional blocks introduces constraints related to the placing of the circuit parts, as some resources are not so copious as others. For instance, the number of CLBs highly exceeds the number of BRAMs and DSPs. For these reasons, the efficiency and design of the FPGA routing has significantly improved from one generation to another. There are mainly two kinds of routing architectures:

- *Mesh-based Routing Architecture*: this design organizes functional blocks as a 2D grid, and interconnects neighbor blocks by means of horizontal and vertical routing tracks and programmable switches.

- *Hierarchical Routing Architecture*: this design organizes functional blocks into clusters, recursively connected by means of wires. The connection between blocks from different clusters passes through different hierarchical levels. This design is particularly suitable for local serial connections, like in dataflow streaming applications.

A dedicated interconnection network is in charge of managing clock signals. Also in this case, the efficiency of the clock network is critical, as it has to guarantee low skew between all the registers from the same clock domain.

2.2.5 FPGA Configuration

The FPGA configuration consists in setting the bits of a set of *configuration points* according to the design to implement on the device. In particular, the configuration points include the truth table of the LUTs, the select signal of the CLB multiplexers, the initial state of the D FFs, the interconnection network, and so on. The result of the configuration is the circuit designed by the user, which connects the different functional blocks on the FPGA to perform the target computation.

The *bistream* file, a binary file in a vendor-specific format, configures the FPGAs mapping its content to the programmable bits. The bitstream file is the final result produced by vendor-specific tools as the end of the FPGA design process. SRAM or flash memories are the most common methods to store the configuration bitstream in FPGAs.

The FPGA configuration may be complete or partial. Indeed, while a complete configuration sets all the programmable bits of the FPGA, partial configuration permits to configure just a portion of the FPGA. In particular, *partial dynamic configuration* is the process of partially configuring the FPGA while maintaining the rest of the circuitry operating without interruption. This process increases the flexibility of the FPGA and permits to design modular circuits.

2.2.6 FPGA Design Process

The high complexity of typical FPGA designs makes the circuit design unfeasible manually. For this reason, hardware designers rely on sophisticated Electronic Design Automation (EDA) tools and closed-source algorithms to guide the transformations from a high-level representation of the design into a low-level hardware specification tailored to the target FPGA technology. During the application of this procedure, vendor tools leverage many abstractions and models, as classified by the *Gajsky-Kuhn Y chart* [56], and then refined by Walker and Thomas [55]. The Y chart, as reported in Figure 2.6, classifies the representations of a circuit under three different domains of description:

- *Behavioral representation* concerns the basic functionality of the circuit,

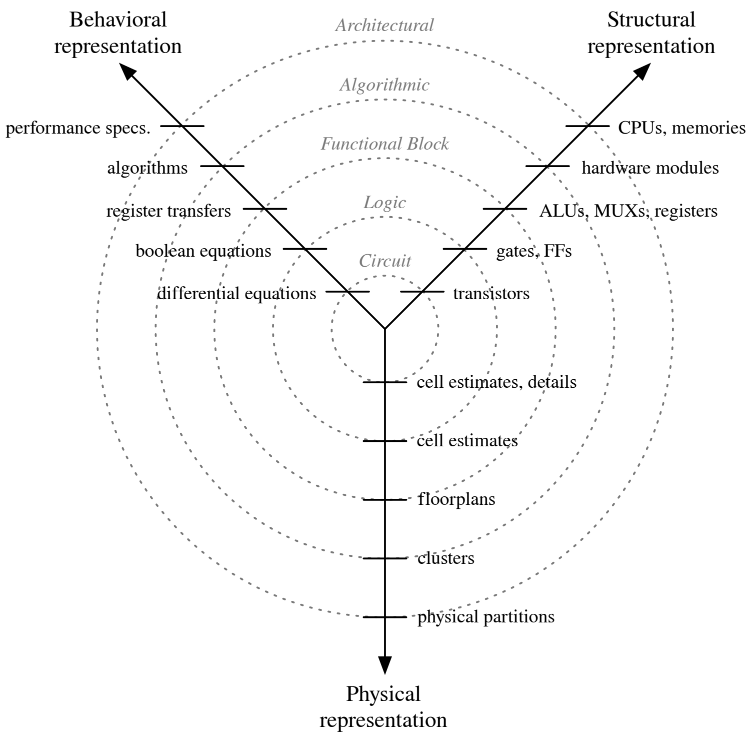

Figure 2.6: *The Y chart. Adapted from [55]. ©1985 IEEE.*

- *Structural representation* is an intermediate domain describing an abstract implementation of the circuit (for instance, a set of components and connections under constraints like area or timing),
- *Physical representation* concerns the physical implementation.

The rings of the Y chart describe multiple levels of abstraction embracing all the domains, so that, as approaching the center of the Y chart, the level of abstraction decreases, while the level of detail increases. A synthesis step describes a transformation moving from a higher level representation to a lower one, or from a domain to another. The complete top-down design synthesis consists in a set of such transformations, starting from the high-level behavioral representation and ending up in the physical one at the center of the Y chart.

Although the Gajski model refers to VLSI circuits, the same principles are also valid for the FPGA design process, the main different consists in the backend implementation and design constraints.

The FPGA design process starts, in most of the cases, from a functional design written in a Hardware Description Language (HDL), like Verilog or VHDL. *High-level synthesis* tools represent an alternative flow, where the hardware designer writes the application in a high-level programming language, like C/C++ or OpenCL, and, from that, the tool generates the HDL code. After the designing part, specialized tools, usually produced by FPGA vendors, synthesize and optimize the circuit, eventually generating the bitstream file, which, in the FPGA context, represents the lowest level of the Y chart. The FPGA design process consists in the following steps:

1. Logic synthesis,
2. Technology mapping,
3. Packing,
4. Placement,
5. Routing,
6. Timing analysis,
7. Bitstream generation.

The purpose of the *logic synthesis* is to convert the HDL description into a netlist, i.e. a set of Boolean gates and FFs. The logic synthesis step involves different technology-independent optimizations, like logic minimization, and algorithms aiming at satisfying the timing and area constraints of the circuit. The goal *technology mapping* step is to find a mapping between the original circuit expressed as a netlist and the resources available on the target FPGA, like LUTs, DSPs, and so on. In case of FPGAs equipped with CLBs containing multiple slices, the *packing* step is in charge of forming groups of logic blocks suitable for a direct mapping to a CLB. The task of the *placement* step is to map the logic blocks to the physical regions and resources of the FPGA, in order to minimize the wire-length or maximize the circuit step. After the placement, the *routing* step interconnects the FPGA resources to form the circuit. The *timing analysis* step establishes the speed of the circuit. Finally, the *bitstream generation* step produces the bitstream file to program the FPGA.

Most of the steps towards the final bitstream generation deal with complex optimization problems, which quite often require iterative techniques, like simulated annealing, to converge to an efficient solution. This implies that the Computer Aided Design (CAD) flow is a definitely computation-intensive process, which may take hours or days to synthesize highly complex circuits. Because of this significant difference in the design flow and

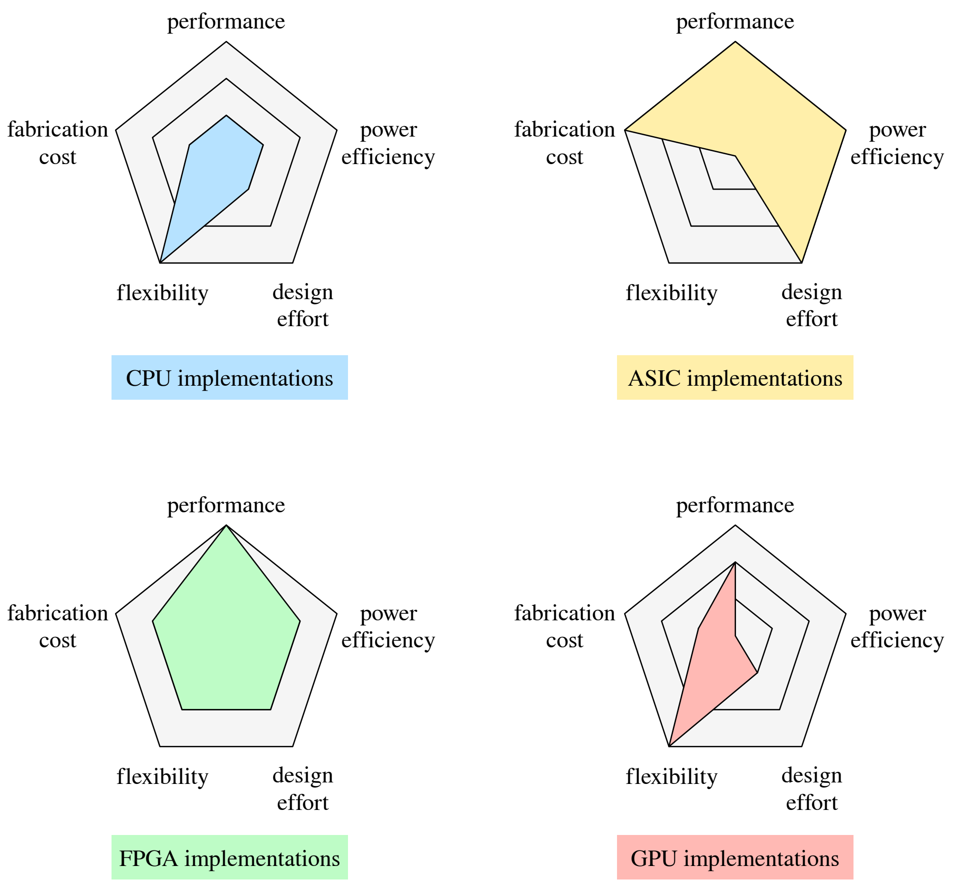

Figure 2.7: *Comparison between hardware architectures. For each indicator, the innermost level means low, whereas the outermost high.*

time with respect to CPU and GPU systems, FPGA tools usually include powerful simulators to perform some stages of verification of the hardware design and analyze its efficiency before the synthesis.

2.3 Architectures Comparison

So far, we have described the architecture of GPUs and FPGAs, which, in addition to ASICs, represent the main devices employed for hardware acceleration. Along with CPUs, each device has its own peculiarities, advantages and disadvantages with respect to the others. We decided to compare each architecture (CPUs, GPUs, FPGAs and ASICs) according to five metrics: performance, power efficiency, design effort, flexibility and fabrication cost. Figure 2.7 displays how each architecture fits the chosen metrics. With respect to the considered indicator, the innermost level of the spider graph means *low*, while the outermost *high*. For instance, CPUs reach an extremely high flexibility at the cost of a relatively low design effort. It is clear that it is not possible to have an architecture able to outperform all the others according to every metric, hence we need to investigate which architecture offers the best trade-off.

With respect to our metrics, the two extreme possibility are CPUs, offering the highest degree of flexibility but limited achievable performance, and ASICs, achieving the highest performance at the expense of development cost and achievable flexibility. Between these extremes, GPUs and FPGAs, offer good trade-offs between performance and flexibility but both have their advantages and disadvantages. GPUs are a great solution when considering massively parallel repetitive tasks that can be executed in a Single Instruction Multiple Data (SIMD) fashion. In addition, GPUs are flexible and easy to program using high level languages and APIs that abstract hardware details. Compared to the fixed hardware architecture of the GPU, FPGAs consist of hundreds of thousands of programmable logic blocks and programmable interconnects that can be configured and reconfigured to deal with control paths as well as data paths. Moreover, the dataflow pattern of an application is exploited in FPGAs through parallelism and pipelining, and designers have the flexibility to trade-off performance for resources. Regarding the disadvantages of these solutions, in GPUs control flow instructions can significantly impact the performance of a program and again the power consumption is pretty high; FPGAs, on the other hand, achieve high performance at an acceptable power consumption but they require a considerable design effort. Given these considerations, FPGAs provide the best trade-off with respect to the other architectures in terms of the chosen metrics.

2.4 Final Remarks

The approaching end of Moore's law and Dennard's scaling pushed researchers to investigate and develop alternative solutions to overcome the limitations in terms of performance and power/energy efficiency of standard general purpose processors. In this context, HSAs represent an effective approach to boost performance while maintaining a relatively low power profile. Indeed, the combination of CPUs with one or more hardware accelerators, like GPUs, FPGAs and ASICs, permits to completely or partially offload compute-intensive applications to the most suitable architecture for that task. GPUs are a good match for highly-parallel and repetitive computations, while FPGAs well fit dataflow applications, and so on. It is critical notice that it does not exist an architecture able to overtake other architectures for each possible computation. Therefore, often the most suitable architecture is the one that offers the best trade-off, according to metrics like performance, energy efficiency, flexibility, and so on.

For this scenario, FPGAs offer the best trade-off with respect to other architectures. Indeed, FPGAs deliver performance higher than CPUs and comparable to GPUs, while consuming definitely less power than these two architectures. On the hard other, even though they can difficulty reach the potentialities of ASICs, FPGAs compensate with their reconfigurable feature, which allows to reconfigure the board both in a static and dynamic way. Despite their benefits, the main limitations of FPGAs consist in their long synthesis times, the complexity in their programming model, as well as the huge design space to explore. This makes FPGAs learning curve highly steep, mainly restricting their usage to experienced hardware designers. In the next Chapter, we will examine in depth the most relevant tools available in literature to develop hardware designs for FPGAs, and will discuss in detail the features of each of those.

CHAPTER 3

How to Program FPGAs

Field Programmable Gate Arrays (FPGAs) are reconfigurable integrated circuits that contain a matrix of functional blocks interconnected together by a switching routing network. Differently from Central Processing Units (CPUs) and Graphics Processing Units (GPUs), which rely on fixed data paths and topologies, the configuration and interconnection of FPGA resources permit to design custom pipelines tailored to a specific application. In this way, users can take advantage of a system able to efficiently implement different functionality, providing a good trade-off between the flexibility of general purpose CPUs and the performance and power efficiency of Application Specific Integrated Circuits (ASICs) [57]. Despite the great opportunities, the fundamental drawback of FPGAs has always been their programmability. The hardware design flow for FPGAs resembles the one available for ASICs, and, historically, the main way to develop hardware design for FPGAs, as well as ASICs, consisted in using Hardware Description Languages (HDLs), especially Verilog and VHDL. These low-level languages permit to describe and define digital multi-signal circuits abstracting the behavior and structure. Many industrial and commercial Electronic Design Automation (EDA) tools, like the ones by Xilinx [58], Synopsys [59] and Mentor Graphics [60], takes in input Register Transfer

Level (RTL) description in Verilog and/or VHDL, and, from that, they perform a sequence of steps towards the generation of the circuit itself. As a disadvantage, in order to efficiently take advantage of these languages, the user requires a significant knowledge and experience in hardware design. Thus, this made the FPGA learning curve definitely steep.

As described in the previous Chapter, the complexity of hardware architectures significantly increased in the last decades, as a single chip contains hundreds of functionalities, and producers release new devices every year. On the other hand, researchers started using FPGAs not only for ASIC-prototyping and networking, but also as hardware accelerators. Nonetheless, the features offered by Verilog and VHDL did not evolve as fast. Hardware designers and EDA tools still use Verilog and VHDL, but limitations like high verification effort and time-consuming and error-prone design make these languages less attractive than how they used to be. As a result, over the last years, new solutions have emerged to cope with the limitations of Verilog and VHDL. In particular, we can identify two main categories: high-level HDLs and High-Level Synthesis (HLS) tools. Modern HDLs offers features and abstractions not available in Verilog and VHDL, like polymorphism, while still providing a design experience close to hardware. On the other hand, HLSs tools permit designers to rely on high-level languages, as C, C++ and OpenCL, to design hardware architectures. Independently from the category, the goal of such tools is to: increase the level of abstraction and productivity for hardware design; permit a high reuse and customization of IPs; reduce verification effort and design errors; make FPGAs accessible to a wider audience of users and developers. As a consequence, these solutions result definitely more appealing than Verilog and VHDL, to both hardware designers who wants to quickly evaluate different architectures, and to application developers who want to hardware accelerate applications, especially High Performance Computing (HPC) ones, and could find significant difficulties in doing that with Verilog and VHDL.

Given these premises, the purpose of this Chapter is to describe the most relevant HDLs and HLS tools available in literature and currently in use.

3.1 Hardware Description Languages

In this Section we analyze the various HDLs available in the state of the art, apart from Verilog and VHDL. For each languages, we describes their features, programming model, as well as their specific advantages and disadvantages with respect to other languages.

3.1.1 ArchHDL

ArchHDL [61] is a HDL for RTL modeling based on C++ developed by the Tokyo Institute of Technology. It is basically a hardware library for C++ that offers features like non-blocking assignments, all the constructs of Verilog, and additional content for simulation optimizations. Designers can simply compile code written using ArchHDL library with a C++ compiler and simulate the hardware design executing the resulting binary. A module description in ArchHDL does not use ports. Therefore, the description about connections between modules is implemented by referring directly to the elements declared in a module. The main focuses of ArchHDL are intuitive module description by object oriented programming and the flexible test-bench description using C++ standard environment.

Even though ArchHDL can generate Verilog for synthesis purposes, its primary objective is to have a faster simulation compared to a Verilog one and also provide easier access to hardware implementation. Users can compile ArchHDL using GCC and parallelize it using OpenMP. This two factors combined permits ArchHDL to achieve significative simulation speedup over Synopsys VCS. Moreover, ArchHDL uses the C++ syntax also to describe test-benches, therefore allowing a fast and easy designs for the latter. Although all the benefits of C++, ArchHDL has some limitations. Because ArchHDL has to be translated in Verilog, all the data types inside the ArchHDL library that can be used to describe Hardware are only the ones that Verilog uses. Some variables, like wires and integer, come from C++ integer type, thus introducing some restrictions, and also ArchHDL supports only arrays with at max two dimensions. ArchHDL supports only one clock signal, and assigns variable only at the positive edge of the clock, thus limiting the possible desings.

3.1.2 Bluespec System Verilog

Bluespec System Verilog (BSV) [62, 63] is a HDL developed by BlueSpec Inc., which aims to provide a general purpose language for hardware design. BSV offers an approach to HDL using *Atomic transactions* to enable high level of parallelism and smoothly refinable designs. Atomic transactions are rules that dictate the behavior of the described hardware. The designer develops modules in BSV, and implements, for each module, both methods and rules. The methods of a module represent the outwards interfaces, while the rules update and modify the internal state of the module. Both rules and methods have guards, and they can fire only if the guards are true and there are no conflicts with respect to that rule. The code in

BSV heavily relies on these transactions to deliver concurrent execution and easy reconfigurability. The designer can set the hierarchical order of the rules without changing the rules themselves, as opposite to System Verilog for instance. BSV carries the dependencies between rules and permits a rule to be executed only if its guard is true and it has no conflicts. Hence, atomicity is also always true in BSV, differently from System Verilog. BSV synthesis tool compiles parallel hardware for the rules, but it always is logically equivalent to a serialized execution of them. This is true at every degree of the program development.

Module interfaces are components of atomic transactions, and derive from C++ and Haskell interfaces. BSV permits polymorphism in order to create easily complex and fully type-checked interfaces in a bottom-up approach by constructing templates. BSV interfaces also support overloading thus simplifying the connections between interfaces. In BSV, designers can easily design components as reusable building blocks and then compose an architecture with those. Moreover, the generation mechanism of micro-architectures supports conditionals, loops and even recursion, making the design process easier and more customizable. On the other hand, the parametrization structure of BSV permits designers to parametrize modules, and, then instantiate specialized versions of the modules in their micro-architectures according to the provided parameters.

BSV claims to be as productive as HLS C-based tools since it allows to easily describe architectures and adapt their modules. Besides, BSV modules can coexist with modules in System Verilog, thus giving the possibility to the developer to use already existing modules.

3.1.3 Chisel

Chisel [64] is a HDL developed by UC Berkeley, embedded in Scala. This choice of Scala offers an easier approach to HDL design compared to Verilog. For instance, the designer can define functions using Scala conventions, construct and nest data structures, design components as classes, redefine operators. Chisel specific libraries permit the designer to also employ specific data types. A key for embedding Chisel in Scala is to support highly parameterized circuits generators, a weakness of traditional HDLs. In this way, designers can declare classes as parameterizable, as well as recursively create hardware subsystems. As another interesting feature, Chisel abstracts the memory representation. The designers can first define it, and then create ports for it. Chisel offers a fast C++ simulator for RTL debugging, as well as a Verilog translator, which permits fine changes and integration with already designed Verilog modules as black-boxes. Nonetheless, is important to notice that Chisel-generated Verilog is slower to simulate on Synopsys VCS than handwritten behavioral Verilog.

3.1.4 Genesis2

Genesis2 [65] is an extension of the functionalities of System Verilog developed by Stanford University. It is built on top of SystemVerilog without modifying its formal syntax. Genesis2 provides hardware designers with a rich software language for writing instructions that specify how to generate modules from a set of input parameters, while the behavioral description is still in System Verilog. Even though it requires System Verilog to describe the hardware modules, Genesis2 uses PERL to express the notion of what hardware to use at a given instance.

In order to allow polymorphism, Genesis2 enables designers to define and give default values to parameters, and then provides a simple mechanism for over-writing these values from external configuration files. Genesis2 permits the implement custom types for the parameters, increasing the flexibility of the modules. Every time the Genesis2 compiler runs, it not only generates code, but also extracts the entire parameterization space, hierarchically, into an XML-formatted description file, thus enabling to easily read out the machine configuration.

3.1.5 HML

HML [66] is an innovative HDL based on the functional language SML. HML aims to be a highly flexible and intuitive HDL language. HML heavily relies on strong type polymorphism permitting to design a function operating over several different data types. However, the designer does not need to specify types; indeed, HML system is in charge of generating I/O behavior and ports. HML enriches SML syntax with specific features for hardware description, including hardware function declaration, signal assignments, bit-vector operations, and extensions for describing both behavioral and structural hardware. In particular, HML defines structures as a set of modules, while behaviors as group of hardware description expressions. Besides, HML does not need to specify clock information, therefore there is no sensitivity list like in VHDL. Because of this, HML supports only one clock cycle. Variables in HML can be declared throughout the program, overcoming a typical restriction of traditional HDLs. Moreover, HML infers the data types, as well as the nature of the variables (e.g. input/output), and, if the type checker cannot find the basic type of a variable, it assigns a special type to the variable, which denotes the variable to have the biggest data width possible. Alternatively, HML offers explicit type definition to increase accuracy. Finally, HML translates the code to VHDL.

3.1.6 JHDL

Just another HDL (JHDL) [67] is a language developed by the Brigham Young University, designed to integrate the host and kernel development. JHDL is integrated within Java as a set of libraries that comprehend both circuit simulation and hardware support. JHDL leverages the object oriented nature of Java language to handle circuits as objects: their resources are created just like Java objects, through the use of class constructors. A single class that wraps all its components and connections represents the whole circuit defined in JHDL. Java methods describes the behavior of the hardware modules, which support parametrization.

The designer can easily select the target of JHDL code execution (hardware or simulation) by just changing the description of the class. This permits a simple host configuration, because the program can be written in one piece and the parts that the designer wants to execute in hardware can be explicitly specified. The simulation at clock level happens by means of JHDL simulation kernel. In order to start a simulation, the circuit class first checks all its components and connection, then issues a clock to start the simulation. On the other hand, if the designer wants to directly run the design in hardware on an FPGA, it is necessary to generate the bitstream first (JHDL produces VHDL code). Nonetheless, in order to permit the translation to VHDL, Java statements that can be used in JHDL are limited. JHDL does not support behavioral synthesis, but provides a graphical schematic viewer and a cycle-based simulator to check the designs. An interesting features of JHDL is the support for contructors/deconstructors in order to reconfigure the circuits on the host side. Since Java does not support explicit object deconstruction (the garbage collector is in change of that), JHDL libraries provide a specific delete method. This permits to simply reconfigure the circuit by means of a method call. JHDL also provides permits that a GUI can be executed from the host while parts of the code are executed on an FPGA.

3.1.7 MaxJ

MaxJ [68] is a HDL developed by Maxeler Technologies. The MaxJ language, a derivative of Java, generates synthesizable code for the Maxeler Dataflow Engine (DFE). The designer employs Java constructs and features to define components and signals (hence, MaxJ supports parametrization and polymorphism), while the syntax remains close to the one from HDL languages. MaxJ offers a great level of abstraction on the operations for the designer, providing also easy integration between host and FPGA, but its use is limited to target Maxeler devices only. MaxCompiler is the specific compiler for MaxJ that compiles the code into output compatible with the FPGA synthesis tools. A key feature of MaxCompiler is the scheduling of the design into a pipelined dataflow architecture. Indeed, starting from MaxJ code, MaxCompiler generates a dataflow graph representing the schedule of the operations. MaxCompiler automatically pipelines each operation, and connects components by means of FIFOs, whose size is inferred implicitly. In this way, MaxCompiler is able to synchronize different paths with different latencies. While being highly optimized for dataflow computations, this also limits MaxCompiler in other applications where the dataflow pattern is not suitable.

The designer is in charge of developing both the host code running on the CPU and the kernel for the FPGA. While the kernel is in MaxJ, the designer may write the host code in different languages, like C/C++, MATLAB, etc. Starting from the kernel description, MaxCompiler automatically generates the interfaces to allow the communication between host and kernel.

3.1.8 myHDL

MyHDL [69] is a language designed by Jan Decaluwe that exploits the Python infrastructure to implement HDL specifications in order to open the hardware development to beginners. Its HDL description is similar to Verilog, but with an easier approach to verification; indeed, it is possible to convert code written in MyHDL into Verilog by means of specific built-in Python libraries and use Python constructors to verificate the designs easily. Also MyHDL supports waveform viewing. MyHDL models hardware as interactive light-weight threads that communicate with each others. In particular, MyHDL description structure is based around *generators*, namely modules that wait for a specific signal in order to perform specific actions. Generators communicate with each other using *generator functions*, a Python feature similar to normal functions but with non-fatal return state. This permits MyHDL to model concurrency efficiently. Moreover, generator functions allow to keep the state of the used functions and resume them if needed, making them usable as ultra-light threads. In this way, it is also possible to pass control information to the simulator. In order to evaluate MyHDL descriptions, a dedicated run-time, called *simulator* is in charge of executing them. MyHDL also supports co-simulation with other HDL simulators by translating MyHDL code into Verilog.

3.1.9 PHDL

PHDL [70] is a Python framework for hardware design, developed by Ali Mashtizadeh at MIT, whose purpose is to significantly increase the level of abstraction for hardware design. PHDL objective does not imply that the design process is faster nor easier, but rather makes the designers more aware of what they are doing. The PHDL framework has two main components: framework classes and a component library. In particular, the library contains pre-made descriptions for the low level components. Designers build components and systems using mainly three types of objects: connectors, components, and connections. Connectors represent actual wires and special collections of wires. PHDL components may be either meta-components or vanilla components. The former are in charge of choosing the best component to employ for a target design. The latter implement the actual logic. Connection objects take care of tying connectors together. Designers implement, on one hand, components and wires as classes through templates provided by PHDL, on the other connections as functions. In addition, PHDL allows to develop and reuse one abstract implementation of a component, thus enabling parametrization of hardware modules. Finally, PHDL outputs Verilog, and it is also possible to integrate PHDL with already existing libraries by creating a wrapper for them.

3.1.10 PyMTL

PyMTL [71] is a highly productive domain specific embedded language for concurrent-structural modeling and hardware design, developed by Cornell University. PyMTL is a framework for Cycle-Level (CL), Functional-Level (FL), and RTL modeling. It relies on Python for the behavioral specification, structural elaboration, and verification of the circuit enabling a rapid code-test-debug cycle for hardware modeling. PyMTL permits a fast access hardware design especially for CL and FL modeling, while RTL modeling resembles more classic HDLs. On the other hand, PyMTL handles the simulation by means of SimJIT, a custom JIT specialization engine that leverages the LLVM compiler and Verilator [72] to automatically generates C++ for CL and RTL models. This permits to speed up the simulation, which, otherwise, would take much longer than the commonly used ones in VHDL. In terms of language features, PyMTL supports dynamic types, providing more flexibility to the possible implementations and simplifying the code and a highly parameterizable behavioral and structural components. Run-time simulation logic relies on nested functions decorated with annotations to indicate their simulation-time execution behavior.

In PyMTL the construction of a module follows a bottom-up approach. At first the designer focuses on the functionality of the algorithm, thus trying multiple implementations of it. PyMTL permits rapid prototyping, because of the nature of the language, and also provides optimizations libraries for this step. Then, after completing and checking the functionality stage, the next step analyzes the operations done within a cycle. The designer can tweak this step to achieve the desired performance, as well as enforce some optimizations provided by specific PyMTL libraries. At last, PyMTL implements the RTL model and generates Verilog. This process enables the designer to start with an implementation of the algorithm from a higher point of view, utilizing the structure of Python for faster prototyping, then to perform a CL inspection to have more information about the timing, and, at last, to work at RTL level to further improve performance. In addition, PyMTL permits the reuse of test-benches created for the CL and FL also for the RTL step.

3.1.11 PyVerilog

PyVerilog [73] is an open source, Python-based toolkit for analysis and code generation of RTL designs developed by the Nara Institute of Science and Technology. PyVerilog provides a lightweight abstraction of Verilog HDL Abstract Syntax Tree (AST) in order to create a design in Verilog by using the AST abstraction and Python. The various tools present inside PyVerilog permit a deep grained analysis of the design, thus providing to the developer numerous ways to optimize the code. In particular, PyVerilog offers code parser, dataflow analyzer, control-flow analyzer, visualizer and code generator for Verilog HDL. The code parser analyzes PyVerilog code and generates an AST based on the preprocessing of the code. From that, the dataflow analyzer constructs a dataflow graph that represents relationships among signals. The dataflow analyzer first passes the AST, builds a list of all the modules within it, and classifies signals (even though it does not analyze assignment statements). Then it checks connections between modules. After that, the dataflow analyzer checks signal assignments. Finally this tool constructs a data flow graph for each signal. The control-flow analyzer generates a graph representation of Finite State Machines (FSMs) in hardware, exploring the previously generated data flow graph. This tool infers the values of candidate conditions from the assignment conditions in the dataflow graph. It also identifies the assertion conditions of the signals, so that it can identify the conditions of state transitions. Finally, the code generator of PyVerilog generates a source code in Verilog from the intermediate representation of an AST written in PyVerilog. PyVerilog deep analysis tools come at a cost, the syntax PyVerilog resembles the Verilog one, but with added complexity, in order to integrate the design with the tools. PyVerilog is also the base for other similar tools. For instance, Veriloggen [74] is a Python open-sourced library designed to generate a Verilog HDL source code constructed on PyVerilog. The purpose of Veriloggen is not to directly design hardware (it generates not synthesizable HDL), but rather to offer abstractions to develop Domain Specific Languages (DSLs) and tools upon.

3.1.12 HDL Comparison Table

Here we summarize the main features of the HDLs described so far. Table 3.1 compares each language in terms of different metrics, like input/output language, support for parametrization and polymorphism, HW/SW codesign option, hardware design simulation, and additional optimizations.

Table 3.1: *Comparison table of the presented HDLs*

HDL Language	Input Language	Output Language	Parametrization	Polymorphism	HW/SW Codesign	Simulation	Optimization
BlueSpec SystemVerilog	Extension of SystemVerilog	System Verilog	Supported	Supported	Not Supported	Supported	For loops Recursion
Chisel	Scala	Verilog	Supported	Supported	Not Supported	Supported	Not Available
MyHDL	Python	Verilog	Not Supported	Not Supported	Not Supported	Supported with other HDL simulators	Not Available
PyVerilog	Python	Verilog	Not Supported	Not Supported	Not Supported	Supported	Not Available
HML	SML	VHDL	Supported	Supported	Not Supported	Supported	Not Available
PyMTL	Python	Verilog	Not Supported	Supported	Supported	Supported	Not Available
Genesis2	Extension of SystemVerilog	SystemVerilog	Supported	Not Supported	Not Supported	Supported	Not Available
ArchHDL	C++	Verilog	Not Supported	Not Supported	Not Supported	Supported	Not Available
PHDL	Python	Verilog	Supported	Not Supported	Not Supported	Supported	Not Available
JHDL	Java	VHDL	Supported	Not Supported	Supported	Supported	Not Available
MaxJ	Java	Not Supported	Supported	Supported	Supported	Supported	For Loops

3.2 HLS

After describing relevant HDLs in the state of the art, we now focus on HLS tools. The purpose of such tools is to further abstract the hardware design process by enabling designers to develop architectures using high-level languages. Here we review the most interesting HLS tools available nowadays. For each of them, we first describe the tool itself, and then we summarize the supported optimizations, as well as the tool perculiar characteristics.

3.2.1 BAMBU

Bambu [75] is a tool developed at the Politecnico di Milano as part of the Panda framework [76]. Its aim is to support the designer in the HLS process of complex applications. Bambu was released in 2012 and, over the past years, Bambu developers have been releasing a new version every year. The input of Bambu is a behavioral description written in C, while the output is a synthesizable RTL implementation in VHDL or Verilog and a test-bench for simulation and validation.

Bambu supports most of the C constructs (function calls, pointers, structs, multidimensional arrays, dynamic resolution of memory accesses) and, since its frontend is built on GCC, it benefits from all the target independent compiler-based optimizations implemented in GCC. Currently, it does not support recursion, but, if necessary, GCC can automatically convert recursive forms into non-recursive forms.

In terms of scheduling, besides the default one, a speculative scheduling algorithm is available. Moreover, the tool gives the possibility to specify in input a fixed scheduling (as XML file) and, as a last option, the designer can activate the post-rescheduling option, useful to better distribute resources.

Supported optimizations: All GCC target independent optimizations, Operation chaining, Pipelining, Resource sharing, Speculation.

Peculiar characteristics: it provides different Pareto Optimal implementations *(tradeoff latency and resources)*; it supports HW/SW partitioning; thanks to its modular organization, it is easily extensible with new algorithms; it efficiently supports complex constructs of C language; it supports different data types.

3.2.2 Catapult-C

Catapult-C [77, 78] is a commercial platform developed by Mentor Graphics to target to both FPGAs and ASICs. It takes C++ or SystemC as input and outputs RTL code in VHDL or Verilog. Both syntax and semantics of C++ are completely preserved, and all basic statements (if, for, while, do, switch) are fully supported. Catapult-C also fully supports functions, pointers and templates. Only two restrictions apply: it does not support dynamic memory allocation/deallocation and it requires the code to be statically determinable, i.e. all properties defined at compile time. On the other hand, Catapult-C supports integer and fixed-points data types, in both cases with arbitrary length. In particular, it manages bit accuracy through templates parameters.

Catapult-C GUI offers a set of built-in graphical analysis tools such as the *Gantt Chart Viewer*, which provides insights on loop profiles, algorithmic dependencies and functional units, or the *Resource Viewer*, to enable full visibility of the HLS results. Finally, Catapult-C also provides a coverage tool to compute quality coverage metrics.

Supported optimizations: Loop optimizations like pipelining, unrolling, merging, Hierarchical Synthesis[1], Scheduling .

Peculiar characteristics: it supports Bit-Accurate data types; native dual-language support; it provides an integrated verification tool; it provides a set of graphical analysis tools to enable easier debug and optimization control.

3.2.3 CoDeveloper

CoDeveloper is a commercial software tool developed by Impulse Accelerated Technologies suited to image and video processing, digital signal processing, and data compression/encryption [79, 80]. CoDeveloper accepts C as input and outputs VHDL or Verilog. The suite includes the Impulse C compiler and an interactive parallel optimizer. In addition, it provides interactive and graphical tools to compile and optimize the code, helping the pipelining and parallelization of critical parts. CoDeveloper natively gives the possibility of using all the standard C development tools to verify and debug, and supplies features like HW/SW partitioning, as well as support for integer and floating point data types.

For the verification part, CoDeveloper suite offers cycle-accurate hardware simulation and testbench generation. It even provides an *Application Monitor* from which the application can be observed during the execution capturing messages, streams of data and any other generated information.

Supported optimizations: Loop optimizations, Scheduling, Pipelining.

[1]Hierarchical Synthesis is the term used in [77] to denote the optimization that generalizes pipelining, allowing different functions to run in a parallel and pipelined manner.

Peculiar characteristics: suited to image and video processing, digital signal processing, data compression/encryption; it supports HW/SW partitioning.

3.2.4 CyberWorkBench

CyberWorkBench (CWB) [81] is an IDE developed by NEC that integrates many tools: from *Cyber Behavioral Synthesis* tool to all their verification and simulation tools built on Cyber. CWB is completely based on C (“All-in-C” approach): all modules in a VLSI design should be described in behavioral C language and all the verification (and debugging) tasks should be done at the C source code level. The main idea of CWB is that a designer should never have the need to debug the RTL code: the IDE even allows the designer to write properties and assertions directly at the C level. CWB geneates Verilog and/or VHDL, and supports legacy RTL and gate net list blocks as black boxes.

It supports FPGA families from Xilinx and Altera and the RTL generated is optimized for the specified technology. CWB offers the possibility to specify the clock frequency and to specify any resource constraints. Among its features, this tool supports both automatic and manual scheduling, is able to handle clocks domain crossing and clock gating, and offers *Quality of Results* reports, with information about area, latency and resources utilized. Finally CWB supplies a *Design Exploration* tool that generates trade-off charts between area, latency and timing, to let the designer choose in full awareness.

The verification tools include an *Automated Test Bench* generator with the ability of generating test benches at the behavioral level and automatically comparing source code results with cycle accurate/RTL simulations.

Supported optimizations: Loop optimizations like merging, unrolling, pipelining, False loop detection, Automatic bit-width optimization, Speculation.

Peculiar characteristics: All-in-C approach; automatic scripts generation to invoke Third Party tools.

3.2.5 DK Design Suite

DK Design Suite [82] is a tool by Mentor Graphics designed for Handel-C language, a rich subset of the C language that targets low-level hardware. DK Design Suite is a complete environment for Handel-C that offers simulation and debugging functionalities. It outputs VHDL, Verilog or EDIF and it supports Altera and Xilinx FPGAs. DK Design Suite offers a *Data*

Streaming Manager that facilitates software partitioning between the processor and the FPGA. Finally, the tool can generate exhaustive report containing information about: errors and warning, block counts, area estimation, summary of the used hardware, optimization.

Supported optimizations: General high level optimizations, Loop optimizations (unrolling), Memory pipelining.

Peculiar characteristics: Integration with leading third party simulators; facilitates partitioning between processor and FPGA; can output EDIF *(Electronic Design Interchange Format)* files.

3.2.6 DWARV

DWARV [83] is a HLS compiler developed at the Delft University of Technology. It is built on CoSy, a compiler development system developed by ACE [84]. Therefore, its characteristics are directly related to the features of CoSy: modular and robust back-end, easiness of extension with new optimizations. Besides, thanks to CoSy, it provides a flexible and easy way to exploit standard and custom optimizations.

DWARV does not restrict the application domain and is able to generate hardware for both streaming and control intensive applications. This tool works on a C subset and outputs VHDL. Over the years, developers extended the supported features of C, including pointers, memory accesses, as well as integer and floating point datatypes. However, DWARV does not support global variables [85], nor recursion and the mathematical functions of the standard C library.

Supported optimizations: Loop optimizations (hoisting, scalar replacement, unrolling, pipelining), Scheduling, If-Conversion, Operation chaining, Bit-Width analysis.

Peculiar characteristics: based on CoSy; it is an academic tool.

3.2.7 eXCite

eXCite [86], developed by Y Exploration, is a synthesis tool for FPGAs and ASICs that takes ISO/ANSI-C as input and synthetizes it to Verilog or VHDL for both Altera and Xilinx FPGAs. It supports both integer and floating point datatypes, and many utilities from `math.h` library. eXCite offers hardware/software partitioning, but communication channels between software and hardware have to be manually inserted. In terms of simulation, this tool provides automatic testbench generation that can be used with any simulation tool to perform the verification, and offers a bit accurate simulation. A relevant functionality is the support for automatic IP reuse and the presence of an IP template library.

Supported optimizations: Generic compiler optimizations, Pipelining (both for loop or entire IP cores), Bit Reduction.

Peculiar characteristics: IP template library.

3.2.8 GAUT

GAUT is an academic and open-source HLS tool dedicated to digital signal processing applications [87–89]. As input, it accepts C and C++ code and it is able to generate VHDL or, in case, SystemC for simulation, visual prototyping or design space exploration. The SystemC models generated are cycle-accurate and bit-accurate and can be employed in external platforms, like SocLib [87]. Thanks to the *Algorithmic CTM* class library from Mentor Graphics, GAUT supports bit-accurate integer and fixed-point variables. GAUT requires to specify constraints on the throughput and the clock period, while the memory mapping and the I/O timing diagram are optional. The final architecture generated by GAUT contains three main components: a processing unit, a memory unity and a communication and interface unit. Finally, GAUT automatically generates a test bench to validate the resulting architecture.

Supported optimizations: Loop optimizations (loop invariant, loop peeling, loop fusion, partial loop unrolling), Operator chaining, Resource allocation optimizations, Bit-Width analysis, Scheduling.

Peculiar characteristics: academic, free and open-source; it outputs SystemC simulation model.

3.2.9 Intel HLS Compiler

Intel HLS Compiler [90, 91] takes untimed ANSI C/C++ as input and generates RTL optimized for Intel FPGAs. It is part of Intel Quartus Prime Design Software for FPGAdesign. Intel HLS Compiler has native support for fixed point and floating point data types, and supports arbitrary width integers. However, the compiler has several limitations regarding the supported subset of C99 and C++. For instance, it does not support dynamic memory allocation, virtual functions, function pointers, and C/C++ library functions except a few math functions. On the other hand, it provides the designer with a tool for design exploration, as well as high level constraints and directives. Moreover, Intel HLS Compiler performs device-specific optimization and technology mapping for Intel FPGAs. This tool also supports verification of the generated RTL by comparison with the original

C++ source model, and is able to generate interactive analysis reports with cross-probing support.

Supported optimizations: Loop optimizations (unrolling, pipelining).

Peculiar characteristics: aimed at Intel FPGAs; well integrated with other Intel tools, like Quartus Prime and Platform Designer.

3.2.10 LegUp

Currently at version 6.1, LegUp is a HLS tool currently developed and released by LegUp Computing [92]. It was first developed, up to the version 4.0, as an open source tool by the University of Toronto [93]. LegUp 4.0 is still available [94] for non-commercial and not-for-profit use. LegUp targets Intel, Xilinx, Lattice, Microsemi and Achronix FPGAs, and the same HLS design can be synthesized for any FPGA vendor.

LegUp environment offers an IDE (LegUp IDE, based on Eclipse), and accepts C and C++ languages as input. Moreover, it supports libraries like pthreads and OpenMP, even though OpenMP is only supported on Linux systems. Starting from the input code, LegUp performs the HLS process producing both RTL in Verilog and report files. The RTL can then be simulated with ModelSim (by Mentor Graphics) and synthesized with vendor CAD tools.

Supported optimizations: Loop optimizations like pipelining and unrolling, Function Pipelining, Bit-Width minimization, If-Conversion, Operator Chaining.

Peculiar characteristics: it is completely vendor-agnostic: the same HLS design can be synthetized for any FPGA vendor; it can synthesizes the program to a hybrid system comprising a processor *(i.e. ARM processor)* and one or more hardware accelerators *(currently on Intel FPGAs).*

3.2.11 ROCCC

ROCCC (Riverside Optimizing Compiler for Configurable Computing) [95, 96] is a C to VHDL compilation toolset particularly suitable for streaming applications. The designer has fine-grained control over the code transformations (to optimize throughput, memory accesses and resources) through an elaborated GUI developed as an Eclipse plug-in. The GUI provides an integrated way to manage the instantiation of modules and cores into C code, control code transformations and optimizations, interface with platforms, and generate testbenches for verification. The same source code can be manually tuned for different FPGA platforms by varying the compiler optimizations in the GUI, without altering the original code. For instance, the designer can manually tune parameters like the number of channels and bitwidth. ROCCC offers support for floating-point and integer operations, while it precludes resource sharing, in favor of better performance.

Supported optimizations: Loop optimizations (unrolling, pipelining, inlining), Pipelining.

Peculiar characteristics: loop optimizations can be specified on a loop-by-loop basis; it supports triple modular redundancy; specific porting to Convey HC machine and Pico devices are available.

3.2.12 SOpenCL

SOpenCL (Silicon-OpenCL) [97] is a tool presented in 2011 able to generate hardware accelerators and System on Chip systems in Verilog, from OpenCL programs. SOpenCL uses C as internal representation language and leverages the transformation from OpenCL to C to coarsen the granularity of the kernel functions. In the second stage the C code is converted in HDL. In this stage the tool uses an architectural template that can be instantiated to match performance requirements and available FPGA resources. During this stage, the available compiler applies different optimizations.

Supported optimizations: Predication, Code Slicing, Modulo Scheduling, Thread Serialization, Variable Privatization.

Peculiar characteristics: it uses the concepts of work-groups (multiple logical-threads) and grid of computation (multiple work-groups). The parallelism granularity is coarsen from a *per-logical-thread* to a *per-work-group* basis; template-based hardware accelerators generation.

3.2.13 Stratus HLS

Released by Cadence, Stratus HLS [98, 99] is a commercial platform that takes C, System C or C++ descriptions and creates RTL implementations for ASIC, SoC and FPGA. This platform integrates Forte Cynthesizer and Cadence C-to-Silicon Compiler into one tool providing full support for designs created with those two tools. It supports industry-standard IEEE 1666 SystemC, C, and C++. Floating point datatypes are available in single and double precision and they are even customizable by the designer.

Stratus HLS provides an IDE that allows the designer to actively choose trade-offs between power, area and performance. The environment provides control and dataflow graphs schematic viewer, pipeline analysis and visualization to evaluate the impact of optimizations. Stratus HLS helps in automating the design and verification flow of hundreds of blocks from transaction-level modeling (TLM) to gates. Moreover, it gives designers synthetizable SystemC building blocks to increase productivity. Finally, Stratus HLS platform also has tools for RTL verification and debugging, power analysis, design exploration, formal equivalence check.

Supported optimizations: loop optimizations, like pipelining unrolling, merging.

Peculiar characteristics: power aware scheduling; accurately identify hotspots in the RTL, both in time and space; other low power optimizations.

3.2.14 Synphony C Compiler

Synphony C Compiler [100] is an HLS tool released by Synopsys that works on C/C++ and outputs RTL code for ASICs or FPGAs. This tool is part of a toolchain offered by Synopsys which includes several different tools for synthesis, design, optimization of power consumption, design exploration, testing and others.

Synphony C offers both streaming and memory interfaces and, by means of pragmas, it allows designers to fine tune performance-related optimization options. Besides, it supplies image processing and streaming libraries This tool supports fixed point arithmetic, but not floating-point operations.

Synphony C offers support for architectural clock gating, both at register-level and at block-level, to enable low power designs. Automatic verification is possibile and is carried out at different levels: after scheduling, after preprocessing, source-code, after synthesis and on the overall generated design. The tool offers automatic generation of testbenches, as well as interactive simulation.

Supported optimizations: Loop optimizations (unrolling, pipelining), Automated single-to-multi-threaded transformations, Hierarchical block-level resource sharing, Timing optimizations for variable bounded loops.

Peculiar characteristics: HDL of other vendors can be directly invoked by the tool; an analysis published by BDTi [101] showed that performance and area metrics for Synphony-produced circuits are comparable with those obtained with AutoESL (that was aquired by Xilinx and became Vivado HLS); it is part of the toolchain offered by Synopsys.

3.2.15 Vivado HLS

Vivado HLS [102], formerly AutoPilot by AutoESL and acquired by Xilinx in 2011, includes a complete design suite that allows to convert high-level languages in HDL. It accepts C, C++, SystemC as input specification languages and can generate hardware in VHDL, Verilog or SystemC. Along with the input code, the designer can provide contraints on the clock period, the clock uncertainty, specify the FPGA target and optimization directives, to better control default behavior of the internal logic.

Vivado HLS provides integer and fixed-point arbitrary precision data types for C and C++, as well as support for the arbitrary precision data types in SystemC. Besides, it supports floating-point operations and provides half-precision floating-point data types.

Vivado supplies tools for verification at the C-level, hardware level, timing and resource usage analysis, and it supports all major simulators in integrated mode.

Supported optimizations: Loop optimizations like unrolling, pipelining, merging, flattening, Operation chaining, Data packing, Function inlining, Bit-Width minimization.

Peculiar characteristics: it is a full-fledged HLS suite; it supports partial reconfiguration; part of a wide toolchain offered by Xilinx.

3.2.16 Tools for Heterogeneous Systems

The following tools have the peculiar purpose of being dedicated to the development of accelerators for heterogeneous systems. This means that these tools are meant to develop accelerators for systems with a CPU and an FPGA (or more than one) where the application runs on the CPU as any standard application, while some functionalities run on the FPGA to enable better performance. The peculiarity of these tools is that the cover the entire hardware design flow, from HLS to bitstream generation.

Intel FPGA SDK for OpenCL

Intel FPGA SDK for OpenCL [103], previously known as AOCL *(Altera SDK for OpenCL)*, is a development environment that enables software developers to accelerate their applications targeting heterogeneous platforms with Intel CPUs and Intel FPGAs. The tool compiles and builds both the kernel and the host from the input code. The compilation can be set to be done in a fast and incremental way. The tool inserts performance counters in the FPGA design and the result obtained can then be reviewed using the Dynamic Profiler tool. Moreover, it offers analysis on the resources and performance, as well as a fast FPGA-based emulation. Once the host application and the kernel match the expected performance, the tool generates a design suitable for deployment, performing a full compilation towards the bitstream.

Supported optimizations: Loop optimizations (unrolling, pipelining).

Peculiar characteristics: what-if kernel performance analysis; fast and incremental FPGA compilation; symbolic debug support; incremental compilation; full deployment supported.

SDAccel

SDAccel [104] is a Development Environment by Xilinx aimed at accelerating functionalities in data centers (such as encryption, search, speech recognition, image recognition) through FPGA resources. It is an Eclipse-based IDE that provides functionalities for code development, profiling and debugging. In addition to that, the IDE provides coding templates and FPGA emulation on x86 platforms.

The SDAccel compiler supports source code using any combination of OpenCL, C, and C++ to develop the hardware kernel. Moreover, SDAccel automatically manages the FPGA runtime. In this way, applications can have multiple kernels swapped in and out of the FPGA during runtime without disrupting the interface between the server CPU and the FPGA. Finally, SDAccel wraps the usage Vivado HLS and Vivado Design Suite toolchains, abstracting and automatizing all the steps towards the bitstream generation.

Supported optimizations: Loop optimizations (unroll, merge, flatten); Array Optimizations (merging, partitioning, reshaping); Pipelining; Bit-Width Minimization.

Peculiar characteristics: It targets acceleration of functionalities of Data Centers; Automation of the complete FPGA design flow; FPGA emulation on x86 platforms.

SDSoC Development Environment

SDSoC [105] provides a framework for developing hardware accelerated embedded processor applications using C and C++ languages. The SDSoC development environment is a tool by Xilinx, and has similarities with SDAccel in their functionalities, even though they target different scenarios. SDSoC comes with an Eclipse-based IDE and compilers for the embedded processor applications. The compiler analyzes the input program to determine the dataflow between software and hardware functions. Then, on one hand, it automatically implements the hardware functions on FPGA, and, on the other, it generates the host code for bare metal, Linux or FreeRTOS. SDSoC provides a completely automated flow for software acceleration on FPGA and system connectivity generation, management of data transfer, synchronization of hardware accelerators, and automatic hardware-software partitioning.

Supported optimizations: Loop optimizations (unroll, merge, flatten); Array Optimizations (merging, partitioning, reshaping); Pipelining; Bit-Width Minimization.

Peculiar characteristics: no prior knowledge of FPGAs is needed; based on the sds++ optimizing compiler; it allows design exploration and system-level profiling.

3.2.17 HLS Comparison Table

The purpose of this Section is to compare the main differences between the HLS tools we analyzed. Table 3.2 summarizes the key features of each HLS tool. For instance, the Table reports the input/output languages, whether the tool is able to generate the bitstream of the hardware design, abstracting the all the steps after the HLS process. The Table also contains other features like support for HW/SW codesign, the type of simulation provided and if the tool can automatically produce a testbench, as well as the target domain and support for floating-point and fixed-point operations.

Table 3.2: *Comparison table of the presented HLS tools*

Tool Name	License	Owner	Input Language	RTL Language	Bitstream	GUI	HW/SW	TestBench	Simulation	Domain	FP	FixP
Bambu	Academic	PoliMI	C	VHDL, Verilog	No	No	Yes	Yes	SW, HW	All	Yes	No
Catapult-C	Commercial	Mentor Graphics	C++, SystemC	VHDL, Verilog, SystemC	No	Yes	No	No	SW	All	No	Yes
CoDeveloper	Commercial	Impulse	ImpulseC	VHDL, Verilog	No	Yes	Yes	Yes	SW, HW	Image/Video Processing	Yes	No
Stratus HLS	Commercial	Cadence	C, C++, SystemC	VHDL, Verilog, SystemC	No	Yes	Yes	Yes	SW, HW	All	Yes	Yes
CyberWorkBench	Commercial	NEC	C, C++, SystemC	VHDL, Verilog	No	Yes	Yes	Yes	SW, HW	All	Yes	Yes
DK Design Suite	Commercial	Mentor Graphics	Handel-C	VHDL, Verilog	No	Yes	Yes	No	HW	Streaming	No	Yes
DWARV	Academic	TU Delft	C	VHDL	No	No	Yes	Yes	HW	All	Yes	Yes
eXCite	Commercial	Y Exploration	C	VHDL, Verilog, SystemC	No	Yes	Yes	Yes	SW, HW	All	Yes	Yes
GAUT	Academic	U. Bretagne Sud	C, C++	VHDL, SystemC	No	No	No	Yes	HW	DSP	No	Yes
LegUp	Commercial	LegUp Computing	C, C++	Verilog	No	Yes	Yes	Yes	HW	All	Yes	No
ROCCC	Commercial	Jacquard Computing	C subset	VHDL	No	Yes	No	Yes	HW	Streaming	Yes	No
Synphony C Compiler	Commercial	Synopsys	C, C++	VHDL, Verilog, SystemC	No	Yes	No	Yes	SW, HW	All	No	Yes
Vivado HLS	Commercial	Xilinx	C, C++, OpenCL SystemC	VHDL, Verilog, SystemC	No	Yes	No	Yes	SW, HW	All	Yes	Yes
Intel HLS Compiler	Commercial	Intel	C, C++	Verilog	No	Yes	No	No	SW, HW	All	Yes	Yes
SOpenCL	Commercial	Univ. of Thessaly	OpenCL	Verilog	No	No	Yes	Yes	SW, HW	All	Yes	No
SDAccel	Commercial	Xilinx	C,C++, OpenCL	VHDL, Verilog, SystemVerilog	Yes	Yes	Yes	Yes	SW, HW	All	Yes	Yes
Intel FPGA SDK for OpenCL	Commercial	Intel	OpenCL	VHDL, Verilog, SystemVerilog	Yes	Yes	Yes	Yes	HW	All	Yes	No
SDSoC Development Environment	Commercial	Xilinx	C, C++	VHDL, Verilog	Yes	Yes	Yes	No	SW, HW	All	Yes	Yes

3.3 Final Remarks

Similarly to what happened with programming languages and frameworks for general purpose processors, tools for FPGA hardware design changed and evolved as well. Modern HDLs and HLS tools offer a new level of abstraction and productivity with respect to Verilog and VHDL. This, on one hand, allows to reduce the design time, facilitate IP reuse, customization and verification, and, on the other, makes FPGA learning curve smoother for non-hardware designers. For instance, tools like Xilinx SDAccel offer an efficient HLS environment, and completely abstract and automatize the hardware design flow, hiding the complexity of the system-level design.

Despite the high level of abstraction provided, the analyzed tools require a certain knowledge and familiarity with hardware design. This is true not only for HDLs but also for HLS tools. The possibility to use high-level languages like C, C++ and OpenCL is for sure an advantage with respect to HDLs, but such languages are not really designed to describe hardware. While HLS compilers make a great job in translating and scheduling high-level languages into hardware RTL, the designer needs to have clear in mind the desired hardware architecture and implement it consequently, in order to facilitate the HLS process. Moreover, many HLS do not automatically optimize the hardware design, but rather they require the designers to use the provided options and directives guide the optimization process. As matter of fact, HLS tools increase abstractions and productivity at cost of control on the resulting design. As a result, also hardware design with HLS tools may become a time-consuming and error-prone task, while still being faster than hardware design with Verilog and VHDL.

Emerging tools, especially domain specific ones, like DSLs and machine learning frameworks, are able to reach relevant levels of both productivity and performance. Indeed, their backends are able to highly optimize the input code thanks to the restriction of the considered domain. On the other hand, tailor-made languages, in case of DSLs, or a high-level language like Python, in case of machine learning frameworks, permit to significantly boost the productivity and express the computation in a simple and intuitive way. Despite the great support for CPUs and GPUs, the support for FPGAs is still limited, even though a combination of such tools with modern HDLs and/or HLS tools could provide a great benefits to designers and further lower the bar for FPGA design. To this end, the next Chapters will describe the main contributions of this work: a framework to automatize the hardware acceleration on FPGA of Convolutional Neural Networks (CNNs), and a common backend to permit multiple DSLs to target FPGAs.

CHAPTER 4

An Automated Framework to Accelerate Convolutional Neural Networks on FPGA

Every day quintillion bytes of data are generated and stored in data-centers of different service provides [106, 107]. These data have different natures: posts on social networks, online transactions, text documents, videos and photos, audio files, GPS signals, and many others. The challenging aspect related to such amount of data (usually called *Big Data*) is definitely their processing and analysis; indeed, this study enables to extract information concealed within the data, like patterns and trends of the market, to name a few. The final outcome may be useful at different ends, ranging from business decisions to research. In order to model and analyze these data, the field of *Big Data analysis* mainly relies on Machine Learning (ML) algorithms (e.g. Neural Networks (NNs), Support Vector Machines (SVMs), decision trees, random forests, etc.). Figure 4.1 reports the most used ML algorithms in data science in 2017 [108]. As a result, many datacenter applications exploit ML to provide end users with services. Well-known examples of these online services are AWS by Amazon [109], the document analysis of Microsoft Bing [110], cloud-assisted voice assistants as Siri [111], Google Photos [112], and so on. However, conventional ML

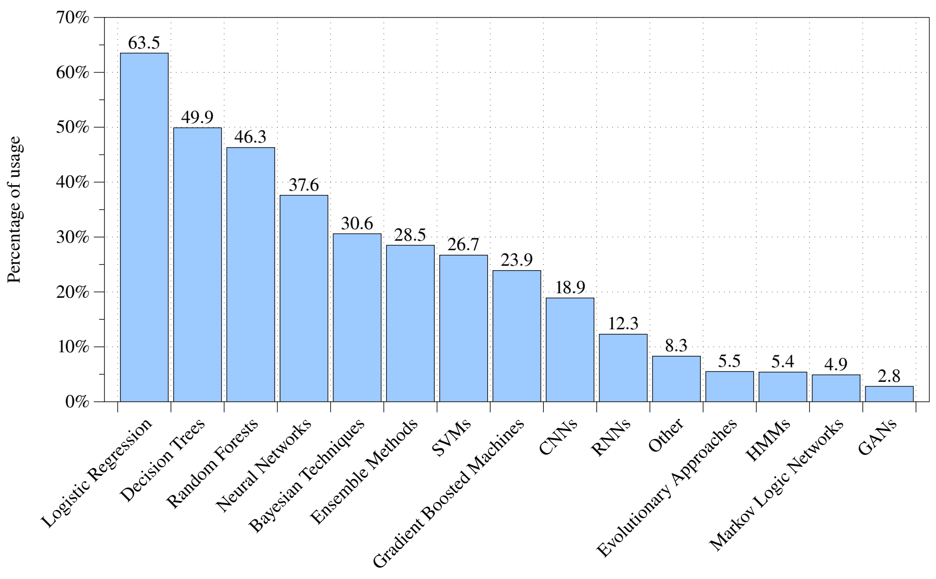

Figure 4.1: *Most used Machine Learning algorithms in data science in 2017 [108].*

have limitations in processing natural data in raw form, and their require a high level of expertise to design feature extractor able to turn raw data into an appropriate representation for ML systems [113].

Moving to a different scale, nowadays computing is going mobile. Technologies have considerably improved from the first mobile phone, and we are now surrounded by a plethora of smart devices, designed to be always connected and always ready to interact with the surrounding environment in real time. This scenario is known as *Internet of Things (IoT)* paradigm, and it describes the interconnection among all kinds of physical objects, ranging from sensors to smart devices. The interaction between IoT devices, as well as their data collection, enabled the rise of new applications in many and different domains, like smart cities [114], healthcare [115], transportation, and more [116, 117]. Just like in the Big Data field, all the information gathered from IoT require an efficient analysis able to uncover hidden patterns within them. Such analysis may occur either remotely in the cloud on the very same device, according to the use cases and user expectations. As a consequence, although energy efficiency remains a critical constraint for these devices, performance steadily gained relevance in the IoT industry. Therefore, it is critical to find the right trade-off between performance and energy efficiency, particularly in the current scenario where new smart devices, like wearable ones, keep entering the market.

In the recent years, *deep learning* [113, 118] has emerged as the most promising solution to face the challenges of the aforementioned fields. Deep learning is a class of techniques composed of multiple layers of representation. The composition of simple but non-linear modules allows to turn a representation into another, increasing the level of abstraction. Deep learning is enabling significant advances in solving problems and research. Indeed, it has been successfully applied to fields like image [118–121] and speech recognition [122–124], bioinformatics [125–127], natural language processing [128], and many others, outperforming other ML techniques.

Among the deep learning class of algorithms, Convolutional Neural Networks (CNNs) [129] have emerged as the most effective method for image recognition and classification. This supervised learning algorithm represents a variant of feed-forward NNs, and takes its inspiration from the biological process in the visual cortex of animals. Starting from the raw data of an image, CNNs are capable of both extracting and aggregating more and more accurate features in order to classify the image subject. Thanks to their high level of performance and efficiency, CNNs have quickly become the state of the art algorithm in image recognition and classification [118]. Moreover, in the last years many ML tools and frameworks have emerged thanks to a considerable engineering and research effort. As a result, frameworks like TensorFlow [35], Caffe [36] and Torch [37] offer efficient solutions to build and train CNNs, as well as other ML algorithms, without a significant expertise of the field.

It is fundamental to notice that, despite all the benefits, constraints in terms of performance and energy efficiency may limit the adoption of deep learning techniques, such as CNNs, at different scales. On one hand, in the datacenter scenario, it is crucial to process and analyze data at a significant high rate, according to the use case requirements. This implies a relevant consumption of power. On the other hand, in the IoT field, the same challenges hold. In addition, the limited resources available on the IoT devices often prevent implementing deep learning techniques on the target hardware [130]. Therefore, since software solutions running on Central Processing Units (CPUs) may not be able to address these challenges, it is necessary to find new solutions able to provide a high level of performance, while maintaining a low power profile.

The computation pattern of CNNs presents a highly repetitive, parallelized and pipelined structure, which makes CNNs an ideal candidate for hardware acceleration. As a consequence, over the last years many hardware implementations of CNNs have emerged in literature, targeting devices such as Graphics Processing Units (GPUs), Field Programmable Gate

Arrays (FPGAs), and Application Specific Integrated Circuits (ASICs) [131–133]. For instance, Google released the Tensor Processing Unit (TPU) [134], a specialized hardware for deep learning algorithms. Among the aforementioned devices, FPGAs offer a good trade-off in terms of performance, power consumption and flexibility. Indeed, although FPGAs may not be able to reach GPUs peak performance for some kind of workloads, their power efficiency makes them preferable to GPUs in case of strict power constraints. On the other hand, even though ASICs offer the best trade-off between performance and power consumption, reconfiguration features allow FPGAs to more easily adapt to changes, without the need of designing a new ASIC. Therefore, FPGAs are an ideal candidate to tackle the challenges described so far. However, the main challenge of FPGA design is the steep learning curve, as well as the long development time needed to generate working and efficient solutions. In the last years, High-Level Synthesis (HLS) tools [102, 104] certainly enhanced productivity and FPGA programmability, allowing hardware designers to leverage high-level languages like C/C++ and OpenCL, instead of Hardware Description Language (HDL), to design custom hardware accelerators. Nonetheless, FPGA design remains a challenging task, and the current level of productivity provided by HLS tools is definitely not close to one offered by high-productivity languages and frameworks. As a result, in case of CNNs and ML in general, end users prefer to rely on the aforementioned ML frameworks, even though they usually target other devices, like CPUs and GPUs. Thus, there is clearly a gap between solid and widely adopted CNNs development frameworks and modern tools to target FPGAs such as HLS ones.

Given these motivations, this work proposes a framework to automated the generation and synthesis of hardware implementation of CNNs on FPGA devices [43,44]. The framework, designed in Python, permits users to provide a high-level description of the network, along with the hyper-parameters of the different layers. Then, after turning the CNN specification into an internal representation based on Google Protocol Buffers [135], the framework produces a C++ implementation of the network, as well as the `.tcl` scripts to automatize all the steps towards the bitstream generation, such as HLS and system level design steps. Moreover, the framework is compatible with industrial ML tools, such as Caffe and TensorFlow. We evaluated the proposed approach on a various set of Xilinx boards, ranging from embedded to high-end devices, as well as on a large experimental set, proving the goodness of the proposed framework on state-of-the-art case studies like USPS [136] and MNIST [137] datasets.

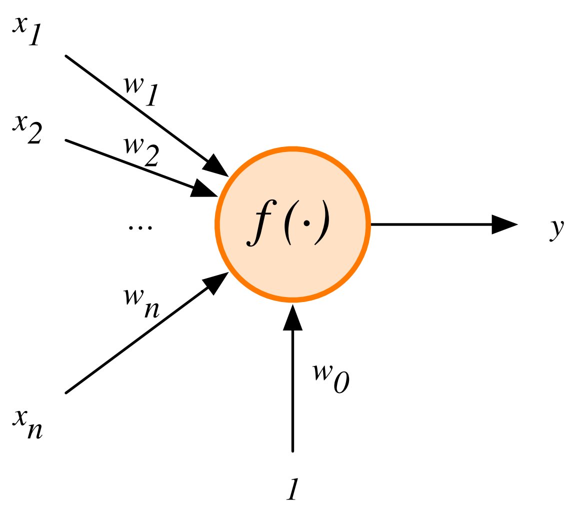

Figure 4.2: *Diagram of the Perceptron model.*

4.1 Background on Convolutional Neural Networks

The purpose of this Section is to present the necessary technical background to understand how CNNs work. First, we introduce the notion of *Perceptron* probabilistic model (Section 4.1.1), then we describe Artificial Neural Networks (ANNs) (Section 4.1.2), and, finally, we provide an overview of CNN topology (Section 4.1.3). Given the goal of the proposed work, i.e. hardware-accelerate the classification process of CNNs, we will mainly focus on that aspect instead of the training one.

4.1.1 Perceptron Classifier

In 1957, Dr. Frank Rosenblatt presented the *Perceptron*, a probabilistic model inspired by the biological neurons in animal brain [138]. As showed in Figure 4.2, the Perceptron model receives an arbitrary number of inputs $x_1, x_2, ..., x_n$ and generates an output y. Each connection between an input value and the nucleus of the Percetron, also known as *synapse*, is weighted with a value. In particular, $w_1, w_2, ..., w_n$ represent the weights of the synapses, while w_0 is a special weight serving as a threshold of the neuron, known as *bias*. After receiving the weighted inputs, the Perceptron applies a function $f(\cdot)$ on them, called *activation function*. There are different types of activation functions, like *Heaviside step*, *sigmoid*, *hyperbolic tangent*, Rectified Linear Unit (ReLU), and so on. The purpose of the activation function is to decide whether the Perceptron can *fire* or not the output. We can describe the Perceptron as follows:

$$y = f\left(\sum_{i=1}^{n}(w_i \cdot x_i) + b\right) \tag{4.1}$$

where b is the bias, defined as $b = 1 \cdot w_0$. According to the activation function, y may assume different values. Let us consider a step activation function. The output of the Perceptron would be as follows:

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{n}(w_i \cdot x_i) + b \geq 0, \\ 0 & \text{if } \sum_{i=1}^{n}(w_i \cdot x_i) + b < 0 \end{cases} \tag{4.2}$$

which basically means that the Perceptron can *fire* if and only if the weighted sum is greater or equal than the bias b.

Provided with a proper set of weights, the Perceptron is capable of discriminating any *linearly-separable* set of inputs. Learning the right set of weights is a task related to the training process of the Perceptron.

4.1.2 Artificial Neural Networks

Despite its features, the main limitation of the Perceptron is its capability of discriminating only linearly-separable patterns. In order to overtake this limitation, a simple yet effective solution consists in connecting multiple Perceptrons to assemble a network of neurons, similarly to the neural networks within the animal brain. The result of such interconnection is an important machine learning algorithm known as Artificial Neural Network (ANN), Neural Network (NN) or Multi-Layer Perceptron (MLP). Figure 4.3 displays an example of ANNs. This kind of network features an arbitrary number of layers of Perceptrons. In particular, we define three categories of layers: *input layer* (the first one), *output layer* (the last one), and *hidden layers* (the ones in the middle). Starting from the input layer, Information pass through the ANN layers and, eventually, reach the output layer, which returns the classification of the input data. Just like in the case of a single Perceptron, the goodness of the inference phase heavily rely on the network training.

Artificial Neural Networks demontrated their capability as classifiers even in image recognition tasks [129, 139, 140]. Indeed, it is quite straightforward to consider an image as a vector of pixels, that become the input vector of a network. Although obtaining promising results, due to the complete interconnection between neurons, NNs are not able to take into

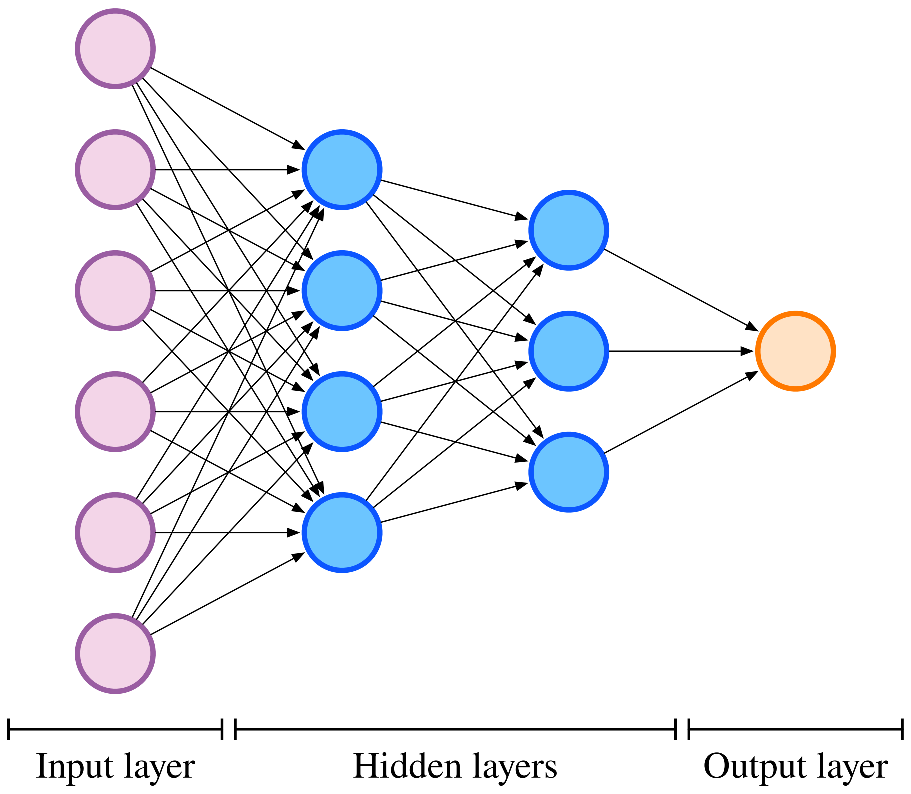

Figure 4.3: *Example of ANN.*

account the instrinsic spatial locality of an image. However, taking inspiration from another biological process, a variant of classic ANNs particularly efficient for image classification has been developed. This new type of neural networks are known as Convolutional Neural Networks (CNNs).

The work in [141], published in the late 1960s, showed that animal visual cortex contains specialized neurons that respond to small regions of the visual field. The same type of cells are present in similar regions across the visual cortex, providing a complete map of the visual space. This working principle lead to the idea behind CNNs structure.

4.1.3 Convolutional Neural Networks

ANNs have demonstrated their efficiency as classifiers in different fields, including the image recognition one [129, 139, 140]. In such a scenario, the image pixels become the input vector of the ANN. However, despite the promising results, ANNs fail in considering the intrinsic spatial locality

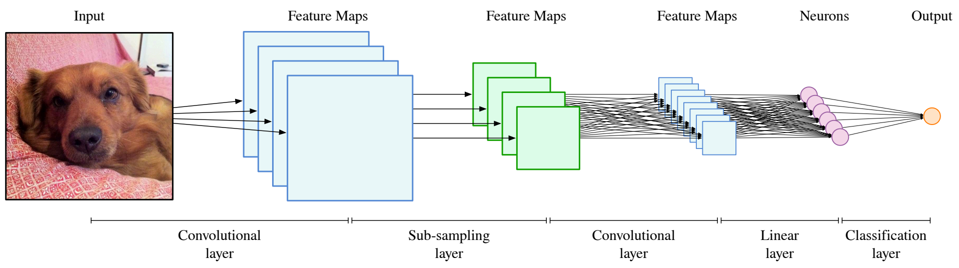

Figure 4.4: *Example of CNN structure.*

of an image. This limitation, mainly due to the complete interconnection between neurons, constrains the image recognition capabilities of ANNs.

To address this issue, in 1990s a new variant of ANN emerged, called Convolutional Neural Network (CNN) [129]. This kind of network takes its inspiration from the animal visual cortex [141], and is specifically tailored to analysis of images and other similar types of data presenting a 2D structure. Figure 4.4 presents the overall architecture of a CNN; it is structured in a configurable chain of layers that can be partitioned in two main stages, called features extractor and classifier, respectively.

Feature Extractor

The features extractor takes in input the image and identifies the various invariant features, such as corners, edges or lines, that are collected in the so-called *feature maps*. This stage exploits the intrinsic spatial correlation in images to define a simpler sparsely-connected organization between adjacent layers: each neuron in a layer takes in input the values produced by a subset of neurons located in contiguous places in the previous layer. The features extractor is composed of a sequence of layers of two different types: convolutional, and sub-sampling layers.

The **convolutional layer** implements the K filters to extract relevant features from the input image; for each filter k, a feature map is generated, as shown in Figure 4.5. Another peculiarity of CNN, implied by the translation invariance property of the features in the image domain, is that all neurons of a single layer share the same filtering function and the same parameters characterization. Therefore, the output feature maps are obtained by repeatedly applying the same filters across sub-regions of the entire image, i.e., by performing a convolution of the input data with each filter function. More technically, the convolutional layer takes as input a 3D volume of dimensions $C \times H \times W$, which are respectively the channels,

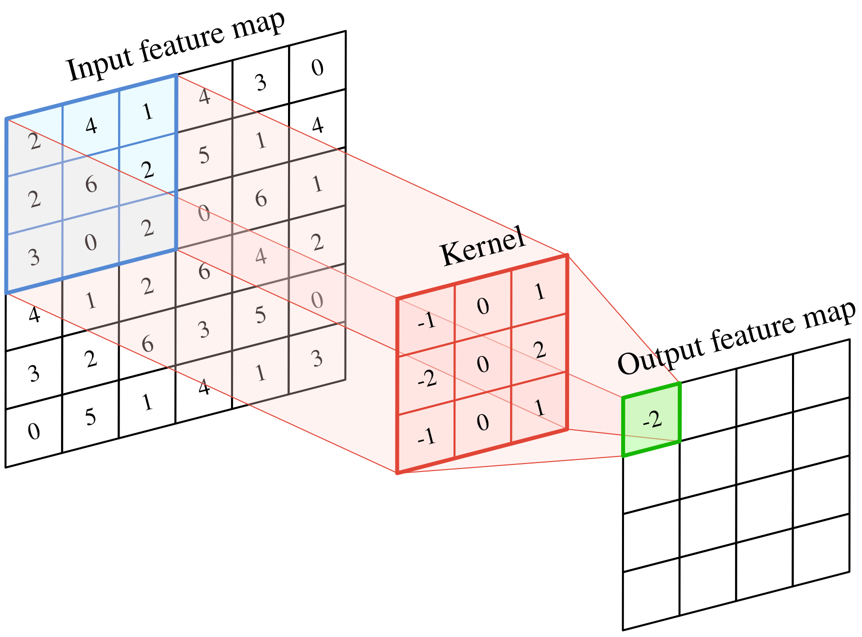

Figure 4.5: *Example of convolution.*

height, and width of the input. This volume represents either the original image or the feature maps produced by the previous layer. For each kernel k, the convolutional layer performs a convolution with the corresponding function represented by a $C_k \times H_k \times W_k$ tensor w of weights (with $C \geq C_k \wedge H > H_k \wedge W > W_k$) and a bias b_k. The following equation describes the convolutional layer works:

$$o^k_{i,j} = \sum_{c=0}^{C} \sum_{h=0}^{H_k} \sum_{m=0}^{W_k} (w^k_{c,h,m} \cdot x_{i+h,j+m,c}) + b_k \tag{4.3}$$

where $o^k_{i,j}$ the output pixel produced by the k-th kernel at coordinates (i, j). It is worth noting that all filters have the same size. Optionally, a nonlinear function, such as $tanh()$ or $max(0, x)$, may be applied on each value in the output volume. Moreover, additional hyper-parameters can be set for configuring the layer. Moreover, two additional hyper-parameters can be set for configuring the layer: (i) the stride S, a positive number, usually set to 1, smaller than the kernel size, with which the filter slides on the input image; (ii) the size P of the zero-padding, which represents the number of zeros that sometimes is convenient to pad the input volume around the border. As result, the generated 3D output volume representing the extracted

```
for (i = 0; i < H - Hk + 1; i++)
  for (j = 0; j < W - Wk + 1; j++)
    for (k = 0; k < K; k++)
      for (c = 0; c < C; c++)
        for (h = 0; h < Hk; h++)
          for (m = 0; m < Wk; m++) {
            w = weights[k][c][h][m];
            x = in[i+h][j+m][c];
            out[k][i][j] += w * x;
          }
```

Listing 4.1: *Pseudo-code of a convolutional layer*

feature maps has a $C' \times H' \times W'$ shape where:

$$\begin{aligned} C' &= K \\ H' &= (H - H_k + 2P)/S + 1 \\ W' &= (W - W_k + 2P)/S + 1 \end{aligned} \tag{4.4}$$

On the other hand, we can easily describe the computation performed by the convolutional layer in an algorithmic way, as reported in Listing 4.1. In this example, the stride S is 1, while the zero-padding P is 0. An important feature of the code is the lack of control structures, which makes the convolutional layer definitely suitable for a dataflow implementation.

The **pooling layer**, also called **sub-sampling layer**, is generally inserted between two convolutional ones to progressively decrease the size of the elaborated data in order to reduce the number of parameters in the kernels and, consequently, the required computations, while forwarding the more relevant features. This layer is again implemented by means of a small kernel (usually 2×2 or 4×4) that is swiped among the analyzed volume in order to cluster locally connected data. The overall structure of the pooling layer is the similar of the convolutional one. The main difference relies in the filtering function. The most common one is the so-called *Max-pooling*, which replaces each submatrix in the input volume with its maximum value. Other types of pooling functions are *Min-pooling* and *Mean-pooling*, which, respectively, extract the minimum or mean value from the current submatrix of the input volume. Finally, in this layer, each channel on the depth axis is considered separately. As a result, the size of the output volume is the following:

$$\begin{aligned} C' &= C \\ H' &= \frac{H}{P_{step}} \\ W' &= \frac{W}{P_{step}} \end{aligned} \tag{4.5}$$

where P_{step} is the *pooling step*, which is usually equal to the stride.

The **activation layer** is an element-wise operator that applies an activation function to every of the input pixels, similarly to what takes place in the Perceptron. The utility of the activation function lets in to *squash* the value of the pixels inside the boundaries of the specific function, keeping off indefinitely growing of the values due to the multiply-accumulate operation in the convolution layer. Besides, such characteristic adds a *non-linearity* that smooths the received classification boundaries. Examples of functions used in the activation layer are the following:

- Sigmoid function: $\sigma(z) = \frac{1}{1+e^{-z}}$ with range $[0, 1]$
- Hyberbolic tangent function: $tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ with range $[-1, 1]$

Classifier

The second stage of the architecture is the **classifier**, that is implemented using a classical fully-connected ANNs. This stage takes as input the feature maps provided by the features extractor, and elaborates them to determine the affinity for the input image with the various considered classes. This stage is composed of a series of linear layers optionally followed by a final normalization operator.

The **linear layer** is composed by J simple Perceptrons, which compute the output values as linear combination of the elements of the input vector opportunely weighted:

$$o_j = \sum_{i=0}^{I} (w_{i,j} \cdot x_i) + b_j \tag{4.6}$$

where x_i is the vector of input elements provided by the previous layer, $w_{i,j}$ the weights and b_j a bias. The number of neurons of the last linear layer is equal to the number of classes to be recognized.

```
for (j = 0; j < OUT_NEURONS; j++)
  for (i = 0; i < IN_NEURONS; i++) {
    w = weights[j][i];
    out[j] += w * in[i];
  }
```

Listing 4.2: *Pseudo-code of a fully-connected layer*

Finally, the last layer in the classifier is a **normalization operator** implemented as a $SoftMax$ operator σ:

$$\sigma_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}} \quad for \;\; j = 1, \ldots, K \tag{4.7}$$

where x_i is the input vector generated by the linear layer. In particular, this operator enforces the K values of the output to lie in range $[0, 1]$ and to sum up to 1, such that they can be interpreted as the probability of the input to belong to a certain class (i.e. the maximum probability).

Similarly to the convolutional layer, the computation pattern of fully-connected layers does not contain any control structure, which makes it definitely fitting for hardware acceleration. Listing 4.2 shows the pseudo-code of a fully-connected layer.

4.2 Proposed Framework

In this Section we describe in detail the proposed framework. As stated in the previous Sections, over the last years CNNs have been one of the major topics in research and engineering field. As a result, different work focused not only on designing new CNNs and improving accuracy, but also on developing tools and frameworks, like TensorFlow [35] and Caffe [36], to easily implement CNNs and, consequently, increase productivity. However, although many frameworks allow users to efficiently target both CPUs and GPUs, there is still little to no support for FPGAs. As matter of fact, the CNN accelerators for FPGA available in literature are mainly the outcome of manual designs. The implementation of efficient CNN architectures on FPGA is definitely a complex and time consuming task, in particular for developers who are not familiar with hardware design. Thus, a framework able to reduce the development effort of CNNs on FPGA could significantly help this category of developers and widen the adoption of such architecture in different application scenarios.

For these reasons, we developed a framework for the fast-prototyping and deployment of CNN accelerators on FPGA. The goal of the frame-

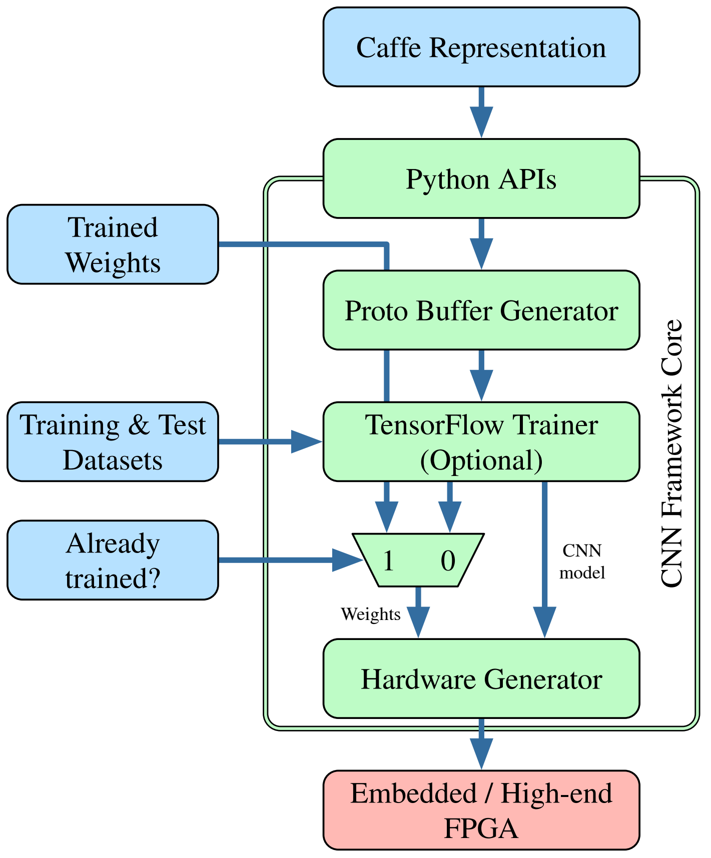

Figure 4.6: *Framework organization and workflow.*

work is to bridge the gap between high-productivity ML frameworks and FPGA design process. The framework automatizes the CNN implementation flow on FPGA, and provides high-level APIs to sketch the network, a C++ library to design dataflow accelerators, as well as an integration with ML framework to train the network. We designed the framework as a set of Python modules. On one hand, we chose Python because almost all the ML frameworks provide Python APIs. This permits to simply interact with other ML toolchains, as we will describe later. On the other hand, the modular architecture of the framework makes it scalable and self-contained; indeed, it is quite straight-forward to use it as a Python package, or employ the single modules as stand-alone Python programs.

Figure 4.6 depicts the overall architecture of the proposed framework. The following three modules compose the core of the framework:

- PROTO BUFFER GENERATOR: this module receives in input the description of the network by means of the framework APIs. The framework supports both a Caffe `.prototxt` definition of a network, and a custom JSON notation. The module converts the input definition into a `.proto` file, i.e. a Google Protocol Buffer [135] file containing a serialized definition of the network.

- TENSORFLOW TRAINER: this module is in charge of training the input CNN via TensorFlow APIs [35]. To this end, the module requires in input both the training and test datasets, as well as the `.proto` definition of the network, which is the outcome of the previous module. The output of the module are the trained weights of the input CNN. It is worth noting that this module is optional; indeed, users can skip the training by supply the trained weights themselves.

- HARDWARE GENERATOR: the last module receives in input a `.proto` file and the weights of the network. The `.proto` not only contains the definition of the network, but also the target FPGA. Starting from those files, the module produces both a C++ implementation of the network suitable for HLS tools and a set of `.tcl` scripts to automatize all the steps towards the bitstream generation.

The next Sections describe in details the implementation of the aforementioned modules.

4.2.1 Proto Buffer Generator

The Google Protocol Buffer [135], also known as *ProtoBuf*, is a flexible, light-weight, extensible mechanism to both store and serialize information in a so-called *message*. ProtoBuf builds the message according to a structured specification of the fields provided by the user. A message field can represent standard data types, like `int`, `bool`, `string`, and enumerations, or another `message` type. After receiving in input the specification, ProtoBuf compiles it and generate ad-hoc APIs to write, read, and manipulate messages as objects. Moreover, ProtoBuf allows to export a human-readable representation of the message in a `.prototxt` file. Thanks to its features, ML frameworks like Caffe [36] take advantage on ProtoBuf as a mechanism to describe a CNN.

In the context of this work, we rely on ProtoBuf as both an *entry-point* to the framework, and as a method to send information from a module to another. In particular, the framework expects in input either a Caffe `.prototxt` model or a JSON file describing the network to accelerate on FPGA. Listing 4.3 describes the structure of the input `message` to the

```
message Project{
    optional string name = 1; // the assigned value
    optional NetParameter network = 2;
    optional Dataset training_set = 5;
    optional Dataset test_set = 6;
    optional Training training_param = 7;

    enum DeviceType{
        ZEDBOARD = 0;
        ZYBO = 1;
        VIRTEX7 = 2;
    }

    optional DeviceType device = 3 [default = ZEDBOARD];
    optional uint32 num_cores = 4 [default = 1];
}
```

Listing 4.3: *Protocol Buffer definition of a `Project` message.*

```
message NetParameter {
    optional string name = 1; // representative name of the network
    repeated LayerParameter layer = 2; //The network layers
    // This field specifies each layer configuration, including
        connectivity and behavior
}

message LayerParameter {
    optional string name = 1; // the layer name
    optional string type = 2; // the layer type
    optional string bottom = 3; // the name of each bottom blob
    optional string top = 4; // the name of each top blob

    // Layer type-specific parameters.
    optional ConvolutionParameter convolution_param = 106;
    optional PoolingParameter pooling_param = 121;
    optional InnerProductParameter inner_product_param = 117;
    optional MemoryDataParameter memory_data_param = 119;
}
```

Listing 4.4: *Protocol Buffer definition the network model, compliant with a subset of the one provided by the Caffe deep learning framework.*

framework. Message `Project` contains all the information necessary to both build the CNN (i.e. `network`) and train it (i.e. `training_set`, `test_set`, `training_param`). Moreover, it provides details about the hardware synthesis, like the target device and the number of cores to instantiate on the FPGA. The tag `optional` on a field indicates that it is not mandatory, while the unique number assigned to each field identifies it in the message binary format [142].

The `NetParameter` message type provides the network model, as described in Listing 4.4, and is compliant with Caffe definition. The content of this message is a sequence of `LayerParameter` fields, which represent the primary types of layers of a CNN, like convolutional, pooling and fully-connected layers. Even though we are considering only a subset of the possible layers, they are enough to build a significant number of state-of-the-art CNNs. Moreover, the scalable definition of the `LayerParameter` message allows to easily introduce additional types of layers.

Listing 4.5 reports the message definitions of the supported types of layers. Each definition provides the hyper-parameters typical of the CNN layer it defines. For instance, the `ConvolutionParameter` message contains fields related to the convolutional layer, like the dimensions of the kernel and the number of output feature maps. On the other hand, the `PoolingParameter` message characterizes the structure of the pooling layer in terms of sub-sampling operator, kernel dimensions, and so on. Then, the `InnerProductParameter` message provides information about the fully-connected layers, namely the number of output neurons. Finally, the `MemoryDataParameter` message defines the dimensions of input image.

The `Dataset` message contains information regarding the input dataset, as reported in Listing 4.6. For instance, it describes the format of the input images. Currently, the framework supports the `IDX` format (for the MNIST dataset [137]) and common image formats like `png, jpg, bmp`.

4.2.2 TensorFlow Trainer

The TENSORFLOW TRAINER module is in charge of proving APIs to train the input CNN and generate the weights of the network, which are fundamental to the inference process. In order to properly train the network, this module requires in input the CNN definition, which comes from the previous module, training and test datasets, and the corresponding labels. The module builds the CNN model according to the ProtoBuf message and maps it into a TensorFlow computational graph definition. At this point,

```
message ConvolutionParameter {
    optional uint32 num_output = 1; // Outputs for the layer
    optional bool bias_term = 2 [default = true];

    repeated uint32 kernel_size = 4; // The kernel size
    repeated uint32 stride = 6; // Defaults to 1
    optional uint32 kernel_h = 11; // The kernel height
    optional uint32 kernel_w = 12; // The kernel width
}

message PoolingParameter {
    enum PoolMethod {
        MAX = 0;
        AVE = 1;
    }

    optional PoolMethod pool = 1 [default = MAX]; // The pooling method
    optional uint32 kernel_size = 2; // The kernel size (square)
    optional uint32 stride = 3 [default = 2]; // Equal in Y, X
    optional uint32 kernel_h = 5; // The kernel height
    optional uint32 kernel_w = 6; // The kernel width
}

message InnerProductParameter {
    optional uint32 num_output = 1; // Outputs for the layer
    optional bool bias_term = 2 [default = true];
}

message MemoryDataParameter {
    optional uint32 batch_size = 1;
    optional uint32 channels = 2;
    optional uint32 height = 3;
    optional uint32 width = 4;
}
```

Listing 4.5: *Protocol Buffer definition of `Layer` messages.*

```
message Dataset {
    optional string path = 1;
    optional string img_file = 8;
    optional string img_ext = 10;
    optional string label_file = 9;

    enum Format{
        MNIST = 0;
        OTHER = 1;
    }

    optional Format format = 2 [default = OTHER];
    optional ImageInfo img_info = 6;
    optional uint32 num_images = 3;
    optional uint32 classes = 7;
}

message ImageInfo {
    optional uint32 channels = 4;
    optional uint32 height = 5;
    optional uint32 width = 6;
}
```

Listing 4.6: *Protocol Buffer definition of a* `Dataset` *message.*

TensorFlow trains the network by means of the *cross-entropy* error function [139]. The user can specify training parameters, like the batch size of images and learning rate, using the `Training` field in Listing 4.3. Naturally, as said in Section 4.2, the execution of the TENSORFLOW TRAINER module is optional, since the user may already have trained the CNN before using the framework. In this case, the user can directly provide the network weights and, consequently, skip this step. Once the training is over, the module exports the weights in either a `csv` or `npy` file format.

4.2.3 Hardware Generator

The purpose of the HARDWARE GENERATOR module is the automatic generation of a hardware implementation of the input CNN suitable for HLS, and its consequent implementation on FPGA. Starting from the ProtoBuf description of the network and its weights, this module produces a C++ implementation of the CNN. In particular, the HARDWARE GENERATOR module implements a hardware module for each layer. Section 4.3 provides more details about the hardware implementation. On the other hand, the module writes the input weights into a header file as static multi-dimensional arrays and initialized. Finally, this module produces `.tcl` scripts to automatize the implementation of the FPGA accelerator. More technically, the `.tcl` scripts create both a Vivado HLS and Vivado project, and run all the necessary steps towards the bistream generation, like the HLS process, the system-level design, and so on. In this way, the user does not have to spend time setting up the tools every time.

4.3 Hardware Design

As stated in the previous Sections, the computation pattern of CNNs is definitely suitable for hardware acceleration on devices like FPGAs, GPUs and ASICs. Indeed, most of the computations within a CNN are basically *dot-products* between two tensors representing, respectively, the weights and the input to that layer. Convolutional and fully-connected layers belong to this category, as we can notice from Equation (4.3) and Equation (4.6). Thanks to the its instrinsic parallelism, dot-product computations well fit specialized hardware. This enables to significantly reduce the latency of such computation with respect to a single multiply-accumulate operation.

Figure 4.7 depicts the hardware accelerator the proposed framework produces from the input CNN definition. The framework implements each network layer as a stand-alone module, which relies on *First-In First-Outs (FIFOs)* to both receive and send data. These design choices permit to build a dataflow architecture able to seamlessly process images in a streaming fashion. As matter of fact, the accelerator forms a *pipeline* whose macro stages correspond to the hardware modules. Internally, each module works in a pipelined way as well. This design guarantees that each pixel has the lowest possible latency inside each module, maximizing the overall throughput of the accelerator. Finally, an AXI Direct Memory Access (DMA) is in charge of retrieving the input images from the off-chip DDR, sending them to accelerator, receiving the results of the classification from the accelerator, and storing them back to the off-chip memory.

We designed the proposed architecture by means of HLS tools, in particular we employed Vivado HLS to sketch it. The framework generates a C++ top function containing calls to templatized functions, which represent each layer of the network. To this end, we created a C++ library for the supported layers. Thanks to the flexibility of C++ templates, we implemented only one function for each layer. The framework customized each call according to the dimensions of the specific layer. Vivado HLS provides directives to guide the synthesis process and help users generate the architecture they have in mind. We took advantage of different directives to optimize the accelerator and tailor it to its goal.

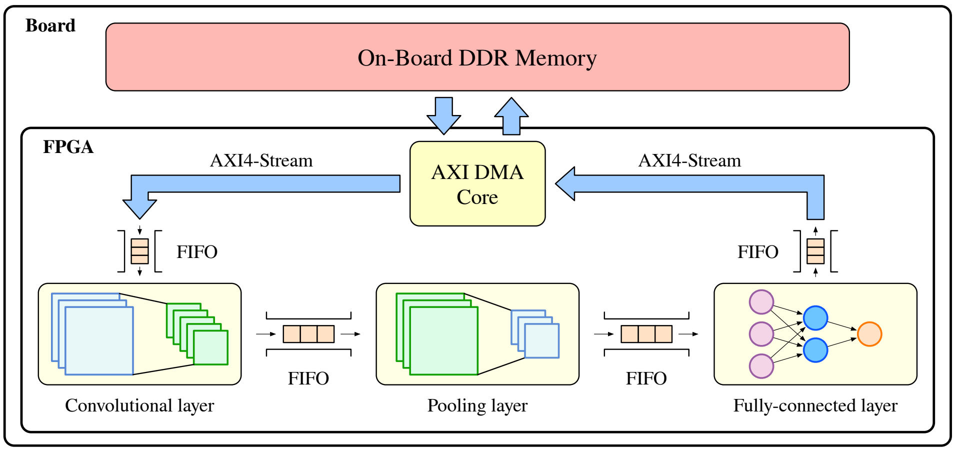

Figure 4.7: *Convolutional Neural Network architectural template.*

First of all, we leveraged the `DATAFLOW` directive and inserted it in the top function. This made Vivado HLS form a pipeline composed of the layer modules. As a result, each module can start computing as soon as it receives the first output data from previous layer. This avoid to stall the computation until the previous module is done and, consequently, increases the throughput. Moreover, each function implementing a layer contains a `PIPELINE` directive to pipeline the loops within it. This permits to reduce the latency of the module itself. Then, the top function directly communicates with the DMA via *AXI-4 stream* interface, which provides a very low overhead for control signals, thus permitting streaming data transfer. More technically, the top function takes two parameters, which represent, respectively, the input and output of the accelerator. We employed Vivado HLS directives to specialize such parameters as AXI-4 streams. Finally, we removed the control signals of the accelerator, which means that it is always active. In this way, as long as there are data in the input FIFO, the accelerator keeps on computing. This permits to seamlessly process batches of images without restarting the accelerator. The next Sections provide additional details about the implementation of convolutional, pooling and fully-connected layers.

4.3.1 Convolutional Layer

The hardware module implementing the convolutional layer leverages the intrinsic parallelism typical of such computation. Indeed, since each filter works independently from the others, we can easily pipeline their execu-

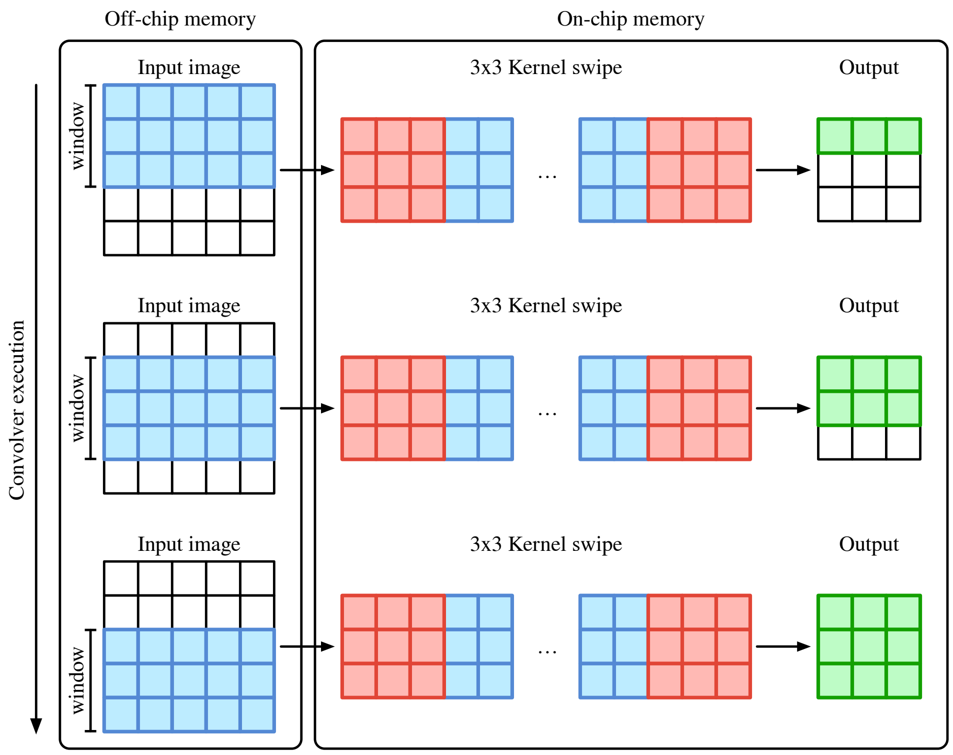

Figure 4.8: *Example of input image loading on on-chip memory.*

tions and produce one pixel per output feature map at each clock cycle. However, in order to apply the filters, the module has to internally store a portion of the input feature maps among different iterations. To this end, each convolutional module exploits shift registers to store the portion of data necessary to compute a row of the output feature maps.

We relied on the `hls::Window` class from the Vivado HLS library to implement the shift registers. This class allocates a bidimensional array of shift registers, and permits to simultaneously access one or more values in the 2D window in the same clock cycle. Besides, the window class permits to shift all the columns left or right and all the rows up or down. We configured the `hls::Window` to store $M \times N$ values, where M is equal to the height of the convolutional kernels K_H and N is equal to $W \times Ch$, respectively the width and the number of the input feature maps. Figure 4.8 depicts the behavior of the window in the convolutional module. Before starting the convolution itself, the module fills the window with the values coming from the previous module. In this way, the module has enough data to start the computation. We pipelined the loop iterating on the output feature maps. As a result, Vivado HLS unrolls the inner loops, which apply the filters on a portion of data within the window. After swiping the filters over the entire window, the module shifts the window content up, removing the first row, which contains the oldest values. The module then loads the next row into the window and the computation goes on until the module produces all the output feature maps.

4.3.2 Pooling Layer

Pooling or sub-sampling layers usually rely on operators, like max or mean operators, to both reduce the size of the output feature maps and forward relevant information to the next layers. The dimension reduction permits to save resources on FPGA, since the size of the window depends on the width of the feature maps.

Similarly to the convolutional module, the pooling module leverages the `hls::Window` to store a portion of the input data, and generates one output per clock cycle in a pipelined way. On the other hand, the stride of the pooling kernel is usually equal to the kernel dimension, hence the window has to shift a proper number of rows before loading the next ones. As soon as the module produces a row of the output feature maps, the module reads new data coming from the previous module, fills the window, and the computation continues.

4.3.3 Fully-Connected Layer

The computational pattern of the modules described so far mainly operates on a relatively small subset of the input values. On the other hand, the fully-connected module requires all the input data to compute each value of the output vector. For this reason, there is no need to use a bidimensional array of shift-registers, also because it would consume a significant amount of logic on the FPGA without a real advantage.

The module reads one input element i per clock cycle in a pipelined manner and calculates its contribution to all the output values j multiplying i by the weights $w_{i,j}$. This requires an additional buffer to store the temporary output values. Besides, Vivado HLS automatically unrolls the inner loop updating the output values. As a consequence, the module may exceed the resources available on the FPGA, in particular the Digital Signal Processors (DSPs) in case of floating point multiplication and addition. Nonetheless, Vivado HLS directives permit to instantiate only a limited number of DSPs. In this way, it is possible to find a good tradeoff between performance and resource usage. Finally, once the computation of the output vector is over, the module writes them in a pipelined way on the output FIFO.

4.3.4 Target FPGAs

So far we have described the CNN accelerator the proposed framework automatically designs. After receiving the network topology, the framework generates C++ files implementing the CNN and a `.tcl` script. This script contains a sequence of commands to automatize the HLS process. Once this process is over, Vivado HLS generates the CNN IP Core. At this point, the framework integrates the CNN IP Core within a system composed of different modules for run-time control and memory management. This step, called *System-Level Design*, is necessary to connect the IP with the off-chip memory and monitor its execution.

According to the target device, the System-Level Design may differ in terms of hardware modules. Currently, the framework supports three different platforms: Zybo [143] and Zedboard [144] platforms, both powered by a chip of the Zynq-7000 All Programmable System on Chip (APSoC) family, composed of a hardwired Arm processor and an FPGA, and the VC707 Development Board [145], powered by a Virtex-7 FPGA. The hardware design for Zynq-7000 devices contains the ZYNQ7 Processing System (a dual-core Arm processor), an AXI DMA module, an AXI Interconnect and the CNN IP generated by Vivado HLS. In particular, the ZYNQ7 Processing System leverages the AXI High Performance slave interfaces (up to 4 ports) to transfer data to the DMA through the AXI Interconnect module. On the other hand, even though the overall structure is quite similar, the hardware design for the Virtex-7 FPGA presents some differences. Indeed, this kind of devices does not contain any hardwired processor like the Arm processor for Zynq-7000 devices. Hence, in order to manage the run-time and communicate with the off-chip memory, we need to instantiate a MicroBlaze soft-processor and a Memory Interface Generator (MIG) module, which provides a standard interface to the memory channels of the on-board DDR. Thanks to this interface, we can transfer data between the off-chip memory and the CNN IP Core. Also in this case, we rely on a DMA module to stream data to/from the CNN IP.

According to the chosen system (Zynq-7000 or Virtex-7), the framework generates a `.tcl` script to automatize the System-Level Design step using Vivado and execute all the following steps towards the bitstream generation. Finally, for both systems, the framework permits to instantiate more CNN IP cores in the design, each with its own DMA module for data transfer.

This enables a *coarse-grained* parallelism for the inference of the given dataset.

4.4 Experimental Results

This Section describes the validation of the proposed framework. We implemented the framework modules using Python 2.7, and relies on Google Protocol Buffers version 2 to send messages from one module to another. We employed the Xilinx Vivado HLx Editions 2016.2 to both design the CNN accelerators and generate the bitstream files.

We implemented different networks, in terms of number of layers and hyper-parameters, for both the U.S. Postal Service (USPS) and MNIST datasets. We evaluated the corresponding accelerators on both embedded and high-end FPGA boards (Zedboard and Virtex-7, respectively). For each dataset, the CNN accelerator performs the inference process using 32-bit floating point operations, while the host collects and normalizes the results by means of a SoftMax operator. We then analyzed each accelerator, synthesized at 100MHz, in terms of execution time, Floating-Point Operations per Second (FLOPS), resource usage and power consumption. In particular, we used the *Energy Logger 4000* by Voltcraft [146] to measure the power absorbed by the target boards.

4.4.1 U.S. Postal Service Dataset

The first case study presents two different CNN models able to recognize the handwritten digits of the USPS dataset, which provides 16x16 greyscale images of digits scanned from the envelops of the U.S. Postal Service [136]. Table 4.1a and Table 4.1b report the architectures of the two CNNs, respectively called *Small* and *Large*. The two networks are quite similar, the *Large* one contains an additional convolutional layer with respect to *Small* one, and the fully-connected layers contain a different number of neurons. The proposed framework trained the two CNN architectures using TensorFlow and generated the C++ implementation of the networks. We evaluated the two networks using 1000 images from the USPS test set and compared the performance of the FPGA designs against a CPU multi-threaded (2 threads) execution on the ARM Cortex-A9 of the Zedboard. Table 4.2a and Table 4.2b report, respectively, the comparison between hardware and software implementations and the FPGA resource usage.

We accelerated the *Small* network on both Zedboard and Virtex-7 board. Due to the limited resources on the Zedboard, a complete parallelization of

Table 4.1: *Architecture of Small and Large CNNs for USPS dataset recognition*

(a) *Small USPS-Net*

LAYER	K_{size}	K_{stride}	INFM	OFM	INDIM	ODIM	FLOP
Conv1	5	1	1	6	16	12	44064
Pool1	2	2	6	6	12	6	864
FC1	–	–	–	–	216	10	4330

(b) *Large USPS-Net*

LAYER	K_{size}	K_{stride}	INFM	OFM	INDIM	ODIM	FLOP
Conv1	5	1	1	6	16	12	44064
Pool1	2	2	6	6	12	6	864
Conv2	5	1	6	16	6	2	19264
FC1	–	–	–	–	64	10	1290

the *Small* network layers was not feasible. To overcome this issue, we reduced the level of parallelism and, consequently, decreased the usage of DSPs and Look-Up Tables (LUTs). Nonetheless, the Zedboard design outperformed the multi-threaded implementation on the Arm processor by a factor of almost 60X in terms of execution time. Besides, even though the Zedboard platform consumed more power when the FPGA is configured, the hardware implementation was more energy efficient than the software one. On the other hand, thanks to the higher number of resources and a different lithography, the implementation of the *Small* CNN on Virtex-7 was able to reach higher performance with respect to Arm processor by means of both fine and course level of parallelism. Indeed, at first, we fully parallelized the *Small* CNN module, and then we instantiated up to four CNN modules on the Virtex-7 board. This permitted to outperform the software implementation by a factor of 98X (one module) and 379X (four modules), and reach peak performance of 11 Giga Floating-Point Operations per Second (GFLOPS). Finally, the four-module design was 40 times more energy efficient than the software implementation running on Arm processor.

For what concerns the *Large* network, it was not possible to implement it on the Zedboard, due to the larger number of resources required with respect to the *Small* one. Hence, we accelerated the *Large* CNN on the Virtex-7 board only. This implementation reached a speedup of 127X compared to the Arm multi-threaded reference. Moreover, the energy consumption of this design is 0.36J, definitely lower than the 4.75J consumed by the software implementation.

Table 4.2: *USPS-Nets performance and resource utilization*

(a) *USPS Hardware vs. Software implementations*

CNN	DEVICE	EXECUTION TIME SW	EXECUTION TIME HW	SPEEDUP	GFLOPS	POWER CPU	POWER DEVICE	ENERGY SW	ENERGY HW
Small	Zedboard	1.67s	0.028s	59.64X	1.76	2.2W	4.40W	3.67J	0.12J
Small	VC707	1.67s	0.017s	98.23X	2.89	2.2W	20.01W	3.67J	0.34J
Small	VC707 (4 Cores)	1.67s	0.0044s	379.54X	11.12	2.2W	21.1W	3.67J	0.09J
Large	VC707	2.16s	0.017s	127.06X	3.83	2.2W	20.9W	4.75J	0.36J

(b) *FPGA resources utilization*

CNN	DEVICE	RESOURCES FLIP-FLOPS	RESOURCES LUT	RESOURCES BRAM	RESOURCES DSP SLICES
Small	Zedboard	31110 (29.24%)	32909 (61.86%)	11 (7.86%)	137 (62.27%)
Small	VC707	54777 (9.02%)	52238 (17.21%)	21.5 (2.09%)	143 (5.11%)
Small	VC707 (4 Cores)	162704 (26.80%)	145684 (47.99%)	66.5 (6.46%)	554 (19.79%)
Large	VC707	223843 (36.86%)	130433 (42.96%)	21 (2.04%)	895 (31.96%)

4.4.2 MNIST Dataset

The second case study focuses on the well-known MNIST dataset [137]. Such dataset provides 28x28 black and white images of handwritten digits. We designed two CNN models based on the MNIST dataset to properly classify the input images. The first network is a custom topology named *MNIST-Net* and contains two convolutional layers with max-pooling and one fully-connected layer. The second CNN is a variant of the *LeNet-5* [129], and is composed of three convolutional layers with max-pooling and three fully-connected layers. Table 4.3a and Table 4.3b summarize the architectures of the considered CNNs.

We relied on the proposed framework to train and implement the two CNNs on FPGA. We evaluated the two networks on the inference of 10,000 images from the MNIST test set. We compared the performance of the FPGA designs in terms of execution time and power/energy consumption against a multi-threaded (4 threads) software implementation running on a Intel Core i7 6700HQ CPU. Table 4.4a compares the performance of the hardware and software implementations, while Table 4.4b reports the FPGA resource usage.

We accelerated both networks on the Virtex-7 board. The FPGA design for *MNIST-Net* was able to process 10,000 images in 0.081s, outperforming the multi-threaded implementation by a factor of 3.33X and reaching 59 GFLOPS. Besides, in terms of energy consumption, the hardware im-

Table 4.3: *Architecture of the two CNNs for inference of the MNIST dataset*

(a) *MNIST-Net*

LAYER	K_{size}	K_{stride}	INFM	OFM	INDIM	ODIM	FLOP
Conv1	5	1	1	8	28	24	235 008
Pool1	2	2	8	8	24	12	4 608
Conv2	3	1	8	16	12	10	232 000
Pool2	2	2	16	16	10	5	1 600
FC1	–	–	–	–	400	10	8 010

(b) *Variant of LeNet-5*

LAYER	K_{size}	K_{stride}	INFM	OFM	INDIM	ODIM	FLOP
Conv1	5	1	1	6	28	24	176 256
Pool1	2	2	6	6	24	12	3 456
Conv2	5	1	6	16	12	8	308 224
Pool2	2	2	16	16	8	4	1 024
Conv3	4	1	16	64	4	1	32 832
FC1	–	–	–	–	64	64	8 256
FC2	–	–	–	–	64	32	4 128
FC3	–	–	–	–	32	10	650

plementation was significantly more energy efficient than the software one (1.62J and 12.38J, respectively). On the other hand, the CNN accelerator of *LeNet-5* was capable of obtaining a speedup of 3.7X with respect to the Intel processor, and, consequently, reaching 62 GFLOPS. Similarly to *MNIST-Net*, the hardware design outperformed the software implementation also in terms of energy efficiency. Finally, it is worth noting that the implementation of *LeNet-5* on FPGA required a high amount of resources, and almost used all the LUTs and DSPs. This was mainly due to the higher number of layers within this network with respect to *MNIST-Net*.

4.4.3 Framework Evaluation

The main goal of the proposed framework is to abstract and automatize the development of a CNN hardware accelerator for FPGA. This permits to simply and speed up a process that, otherwise, could be complex and time consuming. Indeed, the CNN models we implemented count up to more than 9,000 Lines of Code (LoC) tailored to HLS tools. Starting from the network model in a `prototxt` file, the framework generates the CNN C++ design in few minutes, while, starting from scratch, it would take def-

Table 4.4: *Performance and resource utilization of networks for MNIST dataset inference*

(a) *MNIST Hardware vs. Software implementations*

CNN	Device	Execution time		Speedup	GFLOPS	Power		Energy	
		SW	HW			CPU	Device	SW	HW
MNIST-Net	VC707	0.29s	0.081s	3.33X	59	42.7W	20W	12.38J	1.62J
LeNet-5	VC707	0.3s	0.087s	3.7X	62	43.8W	20.5W	13.14J	1.66J

(b) *FPGA resources utilization*

CNN	Device	Resources			
		Flip-Flops	LUT	BRAM	DSP Slices
MNIST-Net	VC707	108995 (17.95%)	131469 (43.3%)	21 (2.04%)	510 (18.21%)
LeNet-5	VC707	258645 (42.6%)	267358 (88.06%)	67.5 (6.55%)	2296 (82%)

Table 4.5: *Summary of the evaluated CNN accelerators*

Network	LoC	FLOPS	FPS	$^{GFLOPS}/_{W}$	$^{FPS}/_{W}$
Small USPS-Net	864	49K	58K	0.14	2.9K
Large USPS-Net	1,583	65K	58K	0.18	2.78K
Mnist-Net	2,300	0.48M	123K	2.96	6.15K
LeNet-5	9,181	0.54M	115K	3.03	5.61K

initely more minutes or hours to get the same result.

Table 4.5 summarizes the implemented networks, and, for each of them, it reports the LoC, throughput and *performance-per-watt* ratio.

4.5 Related Work

In Section 4.1 we described the reasons behind the development of CNNs, as well as their topology. According to the data to classify, it is often necessary to design big networks. This may help increase the level of accuracy of the classification. However, as the size of the network grows, the amount of computations performed by the CNN considerably increases too. The considerable computational demand of CNNs is for sure one of main aspects that makes general purpose processors not suitable for such computation. As matter of fact, modern CPUs fail in delivering high performance when running recent CNNs applications. For this reason, in the last years, many research work evaluated solutions based on dedicated devices such as GPUs, FPGAs or even custom ASIC. As explained in Section 4.1, the computational pattern of CNNs is definitely a good match for hardware acceleration, thanks to the dataflow nature of such computation. In this context, solutions based on GPUs have been able to reach adequate levels of performance [118, 147, 148]. Nonetheless, their high power profile prevent GPUs from being efficiently adopted in scenarios like the embedded and mobile one. On the other hand, ASICs-based solutions [134, 149] are surely appealing in terms of performance and power consumption, but this comes at a price, i.e. lack of flexibility and high production costs. Finally, as shown in different research work ([133, 150–154]), solutions based on FPGAs offer the best trade-off in terms of performance, power consumption, and flexibility with respect to the aforementioned architectures.

The work presented in [131] describes a memory-centric design methodology to accelerate CNNs on FPGA. Thanks to the efficient data pattern access, the authors managed to improve data reuse, and, at the same time, improve performance and energy efficiency. The authors evaluated their designs on a Xilinx Virtex-6 FPGA board, and were able to outperform standard scratchpad memories accelerators.

In [151], the authors leveraged computation reordering and local buffer usage to boost performance and energy efficiency of a CNN accelerator running on FPGA. In addition, the authors described an analytical methodology oriented to the optimization of nested loops for inter-tile data reuse. The experimental evaluation showed a relevant improvement in MicroBlaze soft-core performance, and a 2.1X reduction in terms of data movement.

The work described in [133] takes advantage of the *roofline model* [155], a well-known performance model, to explore the design space of convolutional layers on FPGA. The authors designed the resulting architecture as a configurable single module able to execute all the network layers in sequence. As a result, their implementation of AlexNet [118] managed to surpass the previous implementations available in literature.

ConvNets Processor [156] is a programmable CNN processor. The authors implemented *ConvNets* on a custom board powered by a low-end DSP-oriented FPGA. A *Lush*-based network compiler takes in input a Lush description of the CNN, and generates the instructions for the processor. In [153], the authors designed a CNN accelerator capable of overtaking resource under-utilization issues on FPGA. To this end, the authors developed a modular pipelined architecture structured as a chain of various modules. As a result, the authors were able to optimize the resource usage, and avoid data dependencies between modules. The consequent design, evaluated on a Virtex-7 485T FPGA, reached a 97.1% of dynamic resource utilization, and outperformed the single module state-of-the-art design of the same network by a factor of 1.3X (in terms of throughput).

The work in [154] focuses on a FPGA design for Image-Net large-scale image classification. According to their analysis of literature on FPGA-based CNN accelerators, the authors state that convolutional layers are computational-centric, whereas linear layers are memory-centric. For these considerations, the authors designed a dynamic-precision data quantization method to improve bandwidth and resource utilization. The experimental evaluation showed a definitely small accuracy loss (0.4%), as well as higher performance with respect to previous approaches.

The approaches we have described so far mainly consisted in designing efficient CNN accelerators in terms of performance, power consumption, resource utilization, and so on. The authors manually developed all the aforementioned solutions, and, most of the time, such solutions target a single specific scenario, like the High Performance Computing (HPC) one. As a consequence, re-designing and adapting these architectures to different application scenarios (e.g. the embedded one) easily turns into an error prone and time consuming task. Hence, the goal of the proposed framework is to surpass such limitations and allow users to target different scenarios by means of reusable and automatically generated architectural templates.

4.6 Final Remarks

Convolutional Neural Networks are the state of the art in image recognition and classification. Thanks to their features, they are becoming the standard in many application fields like Big Data Analysis, mobile robot vision, video surveillance and so on. However, the huge amount of data to process makes implementations on modern general purpose architectures impractical, in terms of both performance and power consumption. For this reason, over the last years researchers and engineers have been investigating solutions based on hardware accelerators. In this context, FPGAs proved to be definitely effective as flexible, low-power devices to accelerate the execution of CNNs. Nonetheless, designing efficient FPGA-based accelerators is a time consuming and error-prone task that requires significant experience in hardware design. HLS tools may ease this process permitting to build the hardware accelerator in high-level languages such as C and C++, but still some knowledge in hardware design is fundamental to produce effective designs. This makes FPGA tools not easily accessible for most of the software developers, differently from machine learning frameworks, like TensorFlow and Caffe, which provide high-level tools and abstractions to, almost effortlessly, design CNNs.

For this reason, we developed a framework able to bridge the gap between FPGA hardware design and software development, allowing users to design and implement CNN accelerators on FPGA, starting from a high-level description of the network. In particular, starting from a network definition in Caffe, at first the framework builds an internal representation of the CNN based on Google Protocol Buffer message structure. The framework relies on this representation to make its internal modules communicate. Then, if necessary, the framework trains the network by means of TensorFlow APIs to produce the CNN weights. Finally, the proposed framework generates a C++ implementation of the input network suitable for HLS tools, along with script to automatize all the steps towards the bitstream generation. We evaluated the proposed approach implementing different networks for the USPS and the MNIST datasets. The experimental evaluation showed the efficiency of the resulting hardware designs, which outperformed pure software version running on CPU in terms of both performance and power/energy consumption.

CHAPTER 5

A Common Backend to Target FPGAs from Domain Specific Languages

In the recent years, we have been experiencing an increasing interest in systems composed by multiple heterogeneous architectures. Such systems permit to overcome the limits of homogeneous architectures [157] and thus improve performance while reducing power consumption. For this reason, many high performance systems [23] are currently combining Central Processing Units (CPUs) with architectures like Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), and Application Specific Integrated Circuits (ASICs) to accelerate computations belonging to different fields (like image and signal processing, linear algebra, computational biology, etc.) on the most suitable device for that domain.

Concurrently, many and different tools are emerging in literature to ease and abstract the design of highly parallel applications for such architectures [30, 31]. One of the most interesting solutions in this context is represented by Domain Specific Languages (DSLs). Indeed, current DSLs [32, 33, 47] allow the user to quickly and easily develop portable designs for multiple architectures (mainly CPUs and GPUs). Thanks to the restriction of the domain, DSL compilers are able rapidly explore the design space and deeply

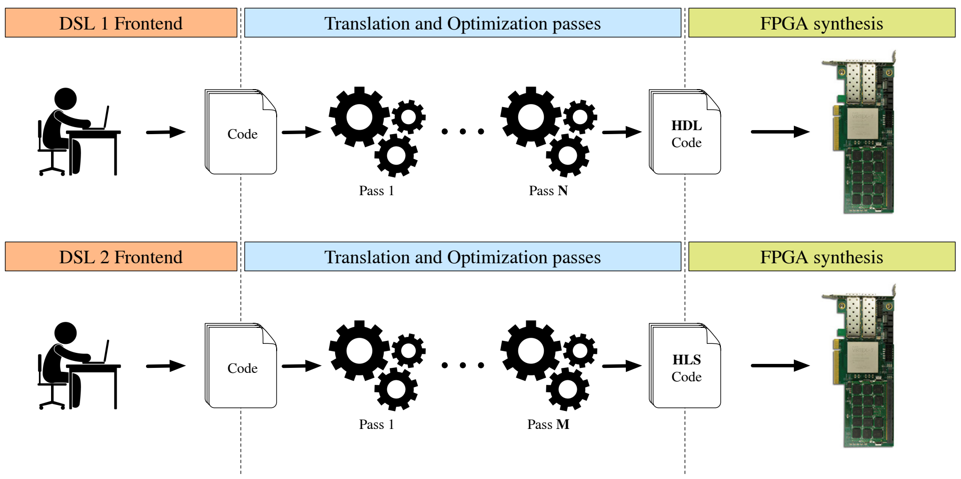

Figure 5.1: *Current design flows of DSLs towards FPGA.*

optimize the resulting implementations. As a result, DSL applications often outperform hand-tuned libraries.

Among the aforementioned architectures, FPGAs currently lack a concrete support for DSLs. Historically, hardware design for FPGAs has always been more complex with respect to the design for CPUs and GPUs, in spite of the great design possibilities FPGAs can provide (for instance, in terms of arbitrary data precision and custom architecture tailored to the target application). Even though in the last years there has been a significant improvement in the toolchain for FPGAs, like High-Level Synthesis (HLS) tools [102] that permit to hardware accelerate algorithms using C/C++ and OpenCL instead of Hardware Description Languages (HDLs) like Verilog and VHDL, the design process remains complex and the supported languages are still limited. As a consequence, there is little to no support for DSLs. In fact, there exist some DSLs able to target FPGAs [38–42], but each one of these DSLs has its own custom flow that, eventually, generates either HDL or code for HLSs tools. Figure 5.1 displays the current design flows towards FPGA available for state-of-the-art DSLs. As a consequence, a common solution to target FPGAs from DSLs is still lacking, which implies that, in most of the times, developers have to start from scratch in order to create an FPGA backend for a new DSL,

This work describes *FROST* [45, 46], a unified backend to efficiently hardware accelerate DSLs on FPGAs. Starting from the an algorithm described in one of the supported DSLs, FROST translates it into its Intermediate Representation (IR), performs a series of FPGA-oriented optimizations steps, and, finally, generates an optimized design suitable of HLS tools. In order to better leverage the features of the FPGA and enhance the performance, FROST provides a high-level scheduling co-language the user can exploit to guide the optimizations to apply, as well as specify the architecture to implement. This allows to easily evaluate different hardware designs and choose the most suitable to the input algorithm.

This work provides the following contributions:

- A common backend capable of supporting multiple DSLs as frontend. In the context of this work, we show FROST integration with Halide and Tiramisu.

- A scheduling co-language to support the user in the generation of an optimized hardware acceleration for the target application, acting on the different levels of the hardware design.

- The FPGA designs generated by FROST obtain performance in line with state-of-the-art FPGA designs of the N-Body Simulation All-Pairs method, and, thanks to a combination of both FROST and Tiramisu scheduling commands, outperform a hand-tuned HLS library by a factor of 17X.

Here the outline of this Chapter: Section 5.6 describes the related work, while Section 5.1 exhaustively analyzes FROST architecture, as well as its high-level scheduling co-language. In Section 5.5 we report FROST experimental evaluation. Finally, Section 5.7 draws the conclusions and gives some insights on the future work.

5.1 FROST Overview

The purpose of this Section is to provide an overview of FROST and its key features. FROST is a common backend for the acceleration of FPGA for DSLs. Given a description of an algorithm in one of the supported DSLs, FROST translates it into FROST IR, and then manipulates and optimized the IR. To this end, FROST provides a high level *scheduling co-language* to allow users to specify the optimizations to apply at different levels (e.g., computation, memory interface, and so on). Once the optimization process is done, FROST generates a C++ code suitable for HLS tools. The final step consists in generating the FPGA bitstream from FROST output. Figure 5.2 gives an overview of FROST design flow.

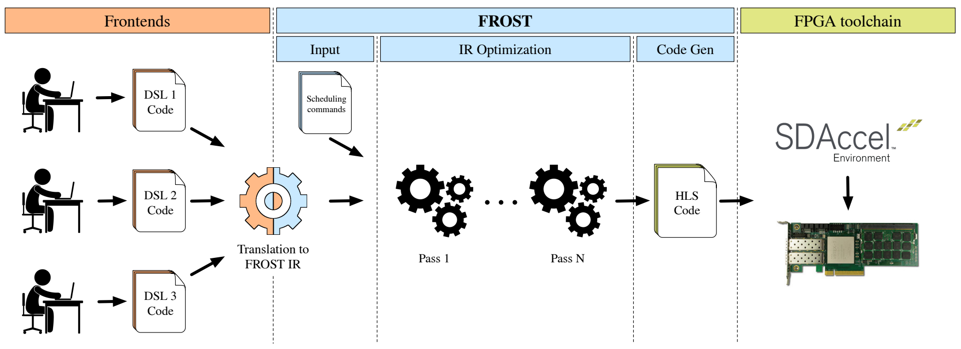

Figure 5.2: *FROST design flow.*

5.1.1 Scope

FROST is designed for expressing data parallel algorithms, in particular algorithms that operate over dense arrays using loop nests. These algorithms are often found in the areas of image processing, dense linear algebra and tensor algebra, stencil computations, and deep neural networks. Moreover, FROST is designed as a common backend for DSLs only. We restrict ourselves to DSLs because DSLs are very effective in producing efficient code for a given target architecture (CPU, GPU, FPGA) because they are restricted to a small set of language features and because their context is limited. Indeed, domain restrictions allow better exploration of the design space and better identification of the computational patterns that are typical in a specific domain. Language restrictions allow better static analysis for the code. For example, many DSLs do not have pointers which allows better static analysis (it is known that static analysis is undecidable if the language has double pointer indirection [158]). As a result, DSLs enable users to easily reach significant performance with a relative small effort with respect to other more general programming languages.

5.1.2 Stack

The overall stack of the proposed work contains three main components: frontends, FROST itself, and an FPGA backend. Figure 5.3 shows the complete stack. While most of the work consisted in designing and implementing FROST and its IR, it was also necessary to efficiently connect FROST with both the supported frontends and FPGA toolchain chosen as backend. To this end, for each frontend, we implemented an ad-hoc translator to FROST IR. At the end of the optimization process, FROST generates code suitable for the target FPGA toolchain, namely Xilinx SDAccel [104].

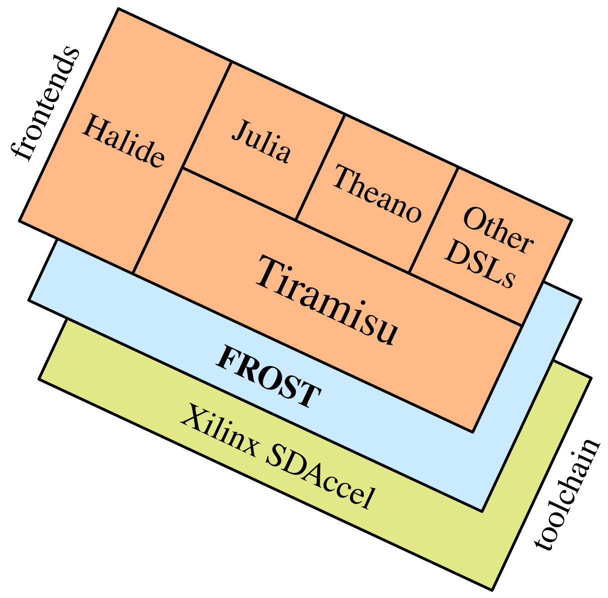

Figure 5.3: *FROST stack.*

In the following Sections, we deeply analyze these three components of FROST stack. In particular, in Section 5.2 we first start describing the DSLs supported as frontends, as well as the rationale behind the choice of designing FROST as a common backend. Section 5.3 focuses on the internals of FROST framework. In particular, it first presents and motivates the FROST scheduling co-language (Section 5.3.1), and then it describes the FROST workflow (Section 5.3.2). Finally, in Section 5.4 we report the FPGA bitstream generation step, which starts from FROST output.

5.2 Frontends

Recently, the use of DSLs and high level languages has been gaining in popularity for many reasons: (1) they provide portability across multiple hardware architectures; (2) they provide high productivity, and (3) they allow the application of certain optimizations such as fusion, and data layout transformations that are difficult otherwise. The input of the FROST backend is the FROST IR which describes the algorithm and a list of optimizations (scheduling) commands to optimize the algorithm. Currently, the FROST IR is fully compatible with Halide [32], a state-of-the-art DSL and compiler for image processing pipelines, as well as Tiramisu [47], a unified optimization framework for DSL compilers, and which presently is integrated in DSLs such as Julia [159] and Theano [160]. In this work, we will focus on presenting FROST itself, and on evaluating FROST with Halide and Tiramisu as frontends. In the next Sections, we describe Halide and Tiramisu.

5.2.1 Halide

Halide [32, 161] is an image processing DSL and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. One of the main features of Halide, with respect to other DSLs for image processing and, in general, pipelines of such computations, is a separate scheduling co-language for expressing when and where to perform the computation. This permits to separate the algorithm from its schedule, enabling the user to write the algorithm only once, and then evaluate different strategies. Indeed, thanks to the scheduling language, the user can optimize the computation at different levels, for instance by applying loop tiling, loop unrolling, vectorization, parallelization and so on. This allows the user to tailor the computation to the algorithm.

Thanks to its features, Halide has quickly become popular and the state-of-the-art DSL for image processing. Indeed, platforms like Google+ Photos rely on Halide. Besides, in the state of the art there are different tools built upon Halide IR. Some examples are Tensor Comprehensions [162] by Facebook, a C++ library and mathematical language to bridge the gap between researchers and engineers, and TVM [163], an optimizing compiler for deep learning.

5.2.2 Tiramisu

Tiramisu [47] is a compiler optimization framework designed for targeting high performance systems. Figure 5.4 portraits an overview of this framework. Tiramisu takes a high level representation of the program (pure algorithm and a set of commands specifying the schedule and data-layout), applies transformations on the representation and generates highly optimized code for the target architectures. Tiramisu is well suited for the implementation of data parallel algorithms (loop nests manipulating arrays). It is designed to hide the complexity and large variety of execution platforms by providing a multi-layer representation suitable for transforming from high-level languages to multicore CPUs, GPUs, distributed machines, and FPGAs, thanks to FROST.

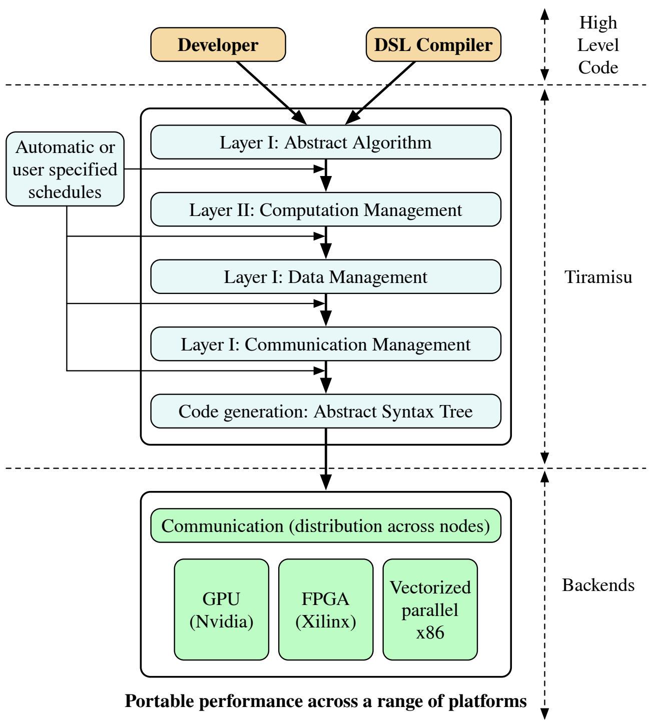
Developer
DSL Compiler
High Level Code
Automatic or user specified schedules
Layer I: Abstract Algorithm
Layer II: Computation Management
Layer I: Data Management
Layer I: Communication Management
Code generation: Abstract Syntax Tree
Tiramisu
Communication (distribution across nodes)
GPU (Nvidia)
FPGA (Xilinx)
Vectorized parallel x86
Backends
Portable performance across a range of platforms

Figure 5.4: *Tiramisu overview.*

Tiramisu allows users to partition their program and specify communication from the same source code using a simple set of scheduling commands. This simplifies programming distributed and heterogeneous systems: the algorithm does not change and only commands that control its execution and communication mapping require modification. In this vision, Tiramisu uses a novel multi-layer IR that fully separates the architecture-independent algorithm from the schedule, data-layout and communication. The multi-layer design makes the algorithm portable and makes it easier to perform each program transformation at the right layer of abstraction. This multi-layer IR also helps Tiramisu address the memory dependence challenge since this design separates data-layout from other transformations.

Tiramisu manages the optimization process of the target application by separating mechanism from policy in scheduling and by removing heuristics and automatic decision-making. This way, Tiramisu allows full control over scheduling while still enabling integration with higher level frameworks for policy-making (deciding which optimization should be applied). Tiramisu guarantees correctness using dependence analysis and thus does not need to impose undue restrictions on its input language to guarantee correctness.

Finally, Tiramisu relies on polyhedral sets to represent the multi-layer IR. This makes it simple for Tiramisu to reason about and implement iteration space and data-layout transformations, since these are represented as transformations on polyhedral sets. It also simplifies deciding about the legality of transformations based on dependence analysis. The polyhedral framework also enables the application of a large set of complex optimizations. Tiramisu does not extend the core of the polyhedral model, but rather it leverages the power of polyhedral compilation to target heterogeneous systems and to generate efficient code that matches kernels from highly optimized libraries such as the Intel MKL library.

5.3 FROST

In this Section, we first describe the motivation for the scheduling co-language, and analyze the commands that the user can express with it (Section 5.3.1). Then, we focus on the IR manipulation and optimization process (Section 5.3.2). More specifically, we explain how the IR changes according to the scheduling commands.

5.3.1 Scheduling Co-Language

We decided to provide FROST with a high scheduling co-language for different reasons. In many performance critical domains, users need code that achieves performance comparable to hand-optimized code. Generating such code requires combinations of non-trivial program transformations that optimization frameworks try to fully automate using cost models, heuristics [164], and machine learning [165]. While automatic optimization techniques provide productivity, they may not always achieve the desired level of performance. A scheduling co-language allows to solve this problem by separating mechanism from policy [1]. This way, FROST allows full control over scheduling while still enabling integration with higher level frameworks for policy-making.

Another reason for the scheduling co-language is the fact that designing efficient architectures for FPGA is definitely a challenging task, and it gets more complex when the goal is to generate them automatically. Again, a set of well defined guidelines for the development of the FPGA design is a viable solution, but it would prevent a fully exploitation of the FPGA fea-

[1] Mechanism means the application of optimizations while policy means deciding which optimization to apply

tures. On the other hand, an exhaustive design space exploration can for sure identify a highly efficient FPGA design, but it would require a significant amount of time. Besides, not even HLS tools do that automatically, but rather they provide the designer with a set of directives to enhance the performance of the hardware design, even though most of the work is still up to the designer's expertise and skills with FPGA design. As a result, producing and evaluating different hardware designs may quickly become a time-consuming and error-prone task.

For the aforementioned reasons, the purpose of FROST scheduling co-language is to (1) separate mechanism from policy and enable users to fully control the generated code; (2) provide the user with a set of possible optimizations in order to tailor the resulting architecture to the input computation, and (3) simplify optimization space exploration. Given the scheduling commands, FROST will automatically generate an optimized version of the input code.

FROST scheduling commands may refer to different aspects of the resulting hardware design. Indeed, they can specify the scheduling of parts of the computation (e.g., loop scheduling), how the data need to be stored on the FPGA memory (either logic or BRAM), or the design of the overall architecture. To this end, we organized the scheduling commands in three different categories (Table 5.1 reports a summary of the scheduling commands currently supported by FROST).

Computation commands: This category contains the scheduling commands that allow to change the scheduling of part of the computations within the overall hardware architecture. In particular, these commands mainly involve the scheduling of loop statements. Indeed, given a function within the design that iterates over the dimensions of the input/output buffers, the user can mark one or more dimensions to be pipelined, unrolled, or (un)flattened. Moreover, the user can choose to vectorize one or more input/output buffers in chunks of `n` bits. This command has an impact on both the access to the off-chip memory and the computation. Finally, the user can require a function inlining.

Local memory commands: This category refers to the scheduling commands related to the data storage on the FPGA. Indeed, given a buffer, such commands enable to partition one or more dimensions of the buffer in a `complete`, `cyclic` or `block` way. According to the command, the data are stored in one or multiple BRAMs, or in logic. In this way, it is possible access in the same clock cycle to different elements of a buffer.

Architecture commands: This last category of scheduling commands impacts on the overall architecture to be generated by FROST. In particular, it defines whether to generate a `tiled` or `streaming` dataflow architecture. In Section 5.3.2 we will better describe the difference between these two architectures. These commands also take care of the communication with the off-chip memory. With respect to that, in case of a target board with multiple DDR memory ports, the user can map an input/output buffer to a specific port. This permits to better exploit the DDR bandwidth and increase performance when dealing with memory bound applications. Currently, FROST provides support for a master/slave communication based on *AMBA AXI4* interface protocol [166]. This protocol allows to either stream or move a tile of data from the DDR to the FPGA local memory and vice-versa. According to the selected command, FROST employs a different approach to move data from/to the off-chip memory.

Finally, it is fundamental to notice that FROST scheduling co-language only focuses on transformations related to the FPGA implementation. Thus, FROST is not designed to perform transformations like loop splitting, loop tiling, and so on. Such transformations are surely useful and necessary in some cases (e.g. vectorization), but, since the supported DSLs, like Halide and Tiramisu, already support them, there was no point in re-implement them also in FROST. Therefore, the idea behind this design choice is that FROST and its frontends has to work in synergy to produce an efficient hardware implementation.

5.3.2 FROST workflow

The input of FROST framework is a set of functions described in one of the supported DSLs, as well as the scheduling commands to optimize the output hardware design. First of all, FROST translates the input functions into FROST IR by means of an ad-hoc translator for each of the frontend DSLs. As a result, FROST represents each function as a data structure that mainly contains: the name of the function itself, its arguments (which are either buffers or scalars), an Abstract Syntax Tree (AST) describing the body of the function, along with other minor parameters. FROST requires the dimensions of the input/output buffers to be specified at this stage of the workflow, otherwise it may not be possible to apply some of the optimizations. At this point, FROST starts applying a series of IR manipulation and optimization steps. In particular, the scheduling commands trigger some of the steps to perform. FROST enforces the scheduling commands in two different ways: IR manipulation or directives for HLS tools. Indeed, some of the commands have a direct map to HLS directives (like loop pipelining, unrolling, etc.), hence FROST inserts them during the code generation.

Table 5.1: *Scheduling Commands*

We assume that `f` is a function, while `m` is the whole computation

Command	Description
Computation scheduling commands	
`f.pipeline(i)`	marks the dimension `i` to be pipelined
`f.unroll(i)`	marks the dimension `i` to be unrolled
`f.flatten(i)`	marks the dimension `i` to be (un)flattened
`f.vectorize(b, n)`	marks the buffer `b` to be vectorized in chunks on `n` bits
`f.inline()`	marks the function `f` to be inlined
Local Memory scheduling commands	
`f.partition(b, d, t)`	marks the buffer `b` to be partitioned on dimension `d` in a `t` way
Architecture scheduling commands	
`m.tiled()`	marks the architecture to be implemented in a `tiled` way
`m.streaming()`	marks the architecture to be implemented in a `streaming` dataflow way
`m.portmap(b, p)`	maps global buffer `b` to memory port `p`

We now focus on the main steps FROST may perform according to the scheduling commands.

Top function generation: The first step of FROST is the definition of the top function, i.e. the main function to be synthesized on FPGA. The purpose of this function is to: (1) invoke the input functions, (2) instantiate the local buffers, (3) declare the memory interfaces, (4) manage the data transfer from/to the off-chip DDR memory. To this end, FROST analyzes the arguments of each function to differentiate the global arguments (i.e., the arguments that refer to buffers to be read/written from/to the off-chip memory) from temporary arguments (i.e. the arguments existing only within the top function). For instance, let us consider a pipeline of two image processing filters, namely `FilterA` and `FilterB`. The arguments/buffers of `FilterA` are `InA` and `OutA`, while the arguments/buffers of `FilterB` are `InB` and `OutB`. Since these two filters work as a pipeline, the output of `FilterA` is the input of `FilterB`, hence `InB` is `OutA`. As a result, `InA` and `OutB` are the global buffers, while `OutA`/`InB` is a temporary buffer. After identifying the global buffers, FROST inserts code blocks to read-

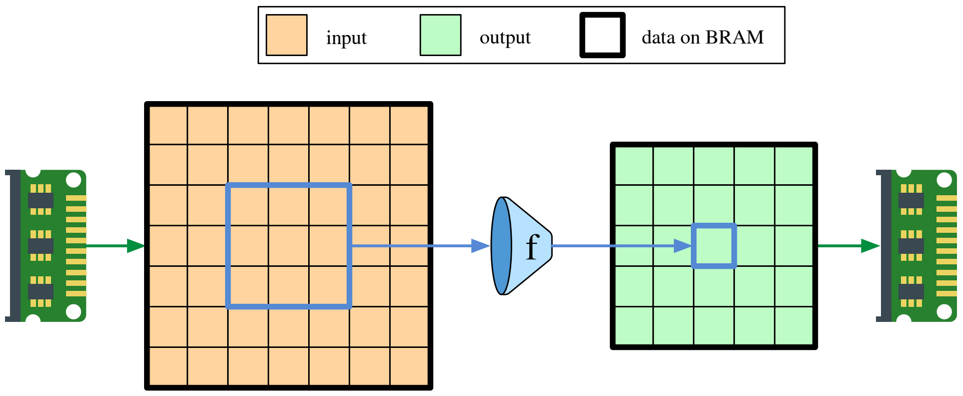

Figure 5.5: *Tiled architecture.*

/write buffers from/to the off-chip memory before/after the computation. More technically, according to the chosen architecture command (namely `tiled` or `streaming`), FROST instantiates different read/write blocks, as well as buffer types.

Tiled architecture: In case of a `tiled` architecture, FROST instantiates the buffers as local arrays, and copies all the data from the off-chip memory, leveraging the memory burst, before starting the computation. Thanks to the data locality, each computation can access data within buffers at different offsets, and iterate multiple times on the same data (ideal for linear algebra kernels). In this case, `partition` scheduling commands may help to improve the performance enabling access to data at different offsets in the same clock cycle. Once the computation is over, the output data are copied back to the off-chip memory. Figure 5.5 portraits the `tiled` architecture.

Streaming architecture: On the other hand, a `streaming` dataflow architecture (more suitable for applications like image processing filters) requires a more complex IR manipulation and analysis of the access patterns within each function. Indeed, to enable a dataflow computation and pipelining between the computations, FROST considers global and local buffers as streams of data (i.e., data FIFOs), and inserts the data coming from the off-chip memory inside such streams. According to the access pattern of the computations, FROST may instantiate line buffers and shift registers to store, respectively, lines of the input (typically an image) and the portion of data to be filtered. This allows to store on the FPGA memory only the data necessary to produce the output. At each clock cycle, the function reads a new element from the input stream and inserts it into the line buffers, while

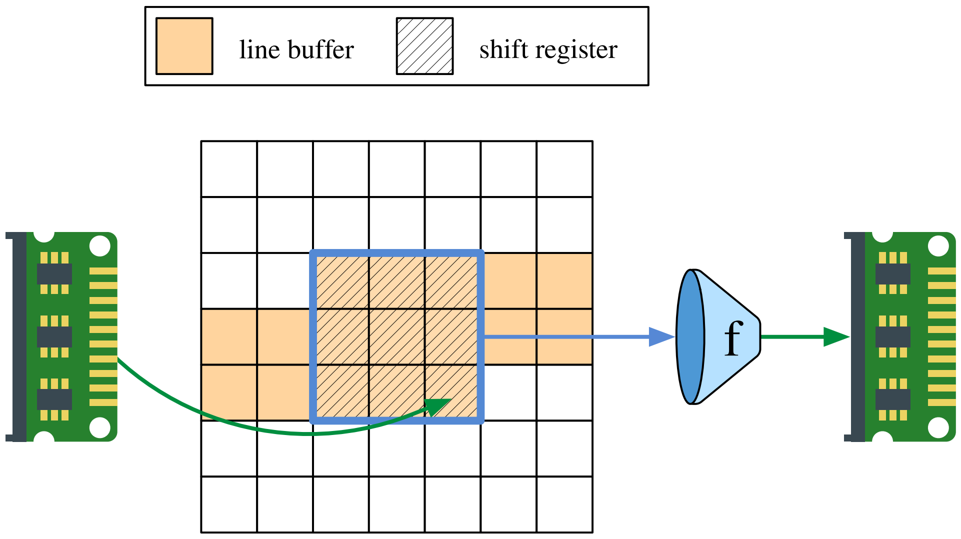

Figure 5.6: *Streaming architecture.*

removing the oldest element from it. At the same time, the function loads data into the shift registers. In this way, as soon as enough data are available within such data structures, the function starts generating outputs and pushing them into the output streams. As a consequence, the functions overlap their execution, significantly reducing the latency of the hardware design. Figure 5.6 displays an overview of the `streaming` dataflow architecture.

Vectorization: Another optimization step that requires significant manipulation of FROST IR is vectorization. Using `vectorization` command, the user marks the buffer data to be packed in bunches of N bits. This command is available both for `tiled` and `streaming` architectures. For instance, a 512-bit vectorization of a 32-bit integer buffer packs 16 integers into a single variable. The vectorization allows to significantly reduce latency of both data transfer and computation itself, but, on the other hand, it implies a significant code restyling. At first, FROST update the data types of the buffers to be vectorized. Then, FROST has to update the access to the data bunches as well. Finally, similarly to the `streaming` architecture, FROST may need to insert shift registers to store a portion of data. In particular, this is necessary when the computation applies a fixed nearest-neighbor pattern to produce the output (e.g., an image processing filter). Hence, FROST analyzes the access pattern to the buffer in order to instantiate a proper sequence of shift registers. Like for the `streaming` architecture, FROST introduces additional code blocks to manage the insertion, access and shift of data within the shift registers. The main drawback of an N-bit vectorization relies in the fact that the number of elements in the buffer (in case of a multi-dimensional buffer, the last dimension) has to be multiple of N/K, where K is the bitwidth of the original buffer data. Consequently, the input may need to be padded, while the output may contain some garbage data. For instance, let us consider a 3×3 filter applied on a $N \times M$ single channel input image. The filtered output should be a $(N-2) \times (M-2)$ image. In case of vectorization, assuming the M-dimension does not need to be padded, the corresponding output is a $(N-2) \times M$ image, where two columns contain garbage data. Padding allows to maintain the hardware design simpler avoiding an invasive control flow.

Final steps: Once the IR manipulation is over, FROST analyzes the new ASTs in order to start the code generation. During the analysis, FROST extracts information related to the to the libraries to include (in case of mathematical functions or particular HLS data structure), and the type of the variables. Basically, FROST builds a lookup table for each function. Finally, FROST visits the ASTs one last time, and, during the generation of the C++ code, enforces the remaining scheduling commands as HLS directives.

5.4 FPGA backend

The last step of FROST workflow consist in generating the bitstream file to configure the FPGA. Once the IR optimization step is done, FROST produces a C++ implementation of the input algorithm suitable for HLS tools. Such code can be immediately imported in a HLS tool like Xilinx Vivado HLS [102] or Xilinx SDAccel [104] to have an estimation of the performance of the current design (for instance, in terms of circuit latency, resource usage, and so on). In this way, the user can verify whether the resulting design reaches, at least theoretically, the expected performance or not, and, if necessary, generate a new optimized design using FROST.

Once satisfied by the produced FPGA design, the user can employ SDAccel to write the host code using SDAccel APIs, and start the synthesis process to, eventually, produce the bitstream file. Indeed, SDAccel environment covers all the steps of the design flow for FPGA (i.e., the HLS and System Level Design steps) and automatizes them. Starting from the kernel generated by FROST, SDAccel first translates it to RTL, and the wraps the resulting IP Core within SDAccel infrastructure. Such infrastructure is in charge of enabling the communication between the board powered by the FPGA and the host machine via PCIe, as well as exploiting partial dynamic reconfiguration enable kernel reconfiguration at runtime. At the end of synthesis process, SDAccel produces the bitstream file. Thus, the user's task consist only in writing the host code using SDAccel APIs to manage the FPGA configuration, and the communication with it via PCIe.

Currently, these steps (namely, evaluation of performance using a HLS tool, host code generation, and SDAccel invocation) are done manually by the user, but we plan to automatize them from FROST, as an additional scheduling command.

5.5 Experimental Evaluation

This Section presents the experimental evaluation of FROST framework. In Section 5.5.1 we describe the experimental setup of our work (Section 5.5.1), and then evaluate the performance of the designs generated by FROST. In particular, in Section 5.5.2 we compare against a hand-tuned library available in Vivado HLS, while, in Section 5.5.3, against state-of-the-art hardware designs for the N-Body Simulation.

5.5.1 Experimental Setup

For each design generated by FROST, we first leveraged Xilinx Vivado HLS 2017.4 to evaluate the performance and resource usage of the design, then Xilinx SDAccel 2017.4 to synthesize it and, consequently, produce the bitstream file. We deployed the final bitstreams on a AWS F1 2x instance, which features a VU9P board powered by a Xilinx Ultrascale+ FPGA. We leveraged SDAccel APIs to manage the application execution, the communication between the host and the FPGA via PCIe, and other aspects of the computation.

5.5.2 Experimental Results: Vivado HLS Video Library

In this Section, we report the evaluation we performed on the designs produced by FROST against the Vivado HLS Video Library, a library implementing several hand-tuned OpenCV functions for FPGA. In this context, we used both Halide and Tiramisu as frontends, and evaluated FROST with 8 image processing kernels, namely `Threshold`, `AddWeighted`, `Erode`, `PaintMask`, `BlurXY`, `Scale`, `Sobel`, `Gaussian`. We decided to target such kernels because they are already available within the Video Library. We designed the first four kernels using Halide, and the remaining with Tiramisu. For each kernel, we compared the resulting design

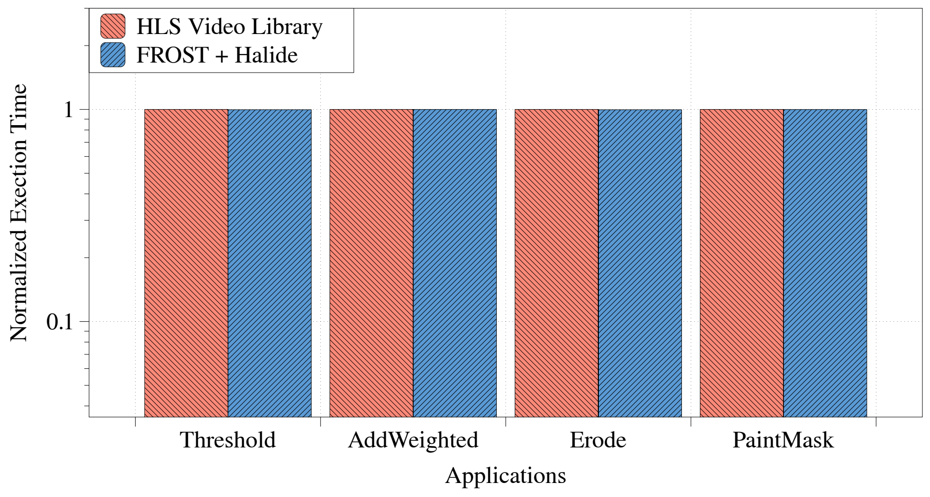

Figure 5.7: *Performance comparison between FROST with Halide and Vivado HLS Video Library.*

(in terms of execution time and resource usage) against the corresponding kernel in the HLS Video Library. The input of each kernel is a 8-bit FullHD (1920x1080) RGB image. The only exception is the `threshold` kernel, which works on single-channel images.

It is important to notice that the Video Library expects the input images arranged in an interleaved format, i.e. a format that encodes each color (and alpha) channel, one after the other, for every pixel (RGBRGBRGB...). The main reason is to maintain consistency with software OpenCV library. This is a good design choice and enables to compute each channel in parallel (i.e. one pixel per clock cycle). However, a different image format, like a planar one, where all of the samples in each channel are stored consecutively and the channels themselves are consecutive in memory as well (RRR...GGG...BBB...), would allow to process more elements in parallel (for instance, N elements belonging to the same channel per clock cycle). This is particularly true when we consider channel-independent filters. In this regard, I demonstrated that FROST is flexible enough to design both interleaved and planar designs, thanks to the integration with its frontends.

It is important to notice that the Video Library expects the input images arranged in an interleaved format, i.e. a format that encodes each color (and alpha) channel, one after the other, for every pixel (RGBRGBRGB...). The main reason is to maintain consistency with software OpenCV library. This is a good design choice and enables to compute each channel in par-

Table 5.2: *Resource usage of Halide benchmarks*

Application	*BRAM_18K* (4320)	*DSP48E* (6840)	*FF* (2364480)	*LUT* (1182240)
FROST				
Threshold	1	0	1851	1196
AddWeighted	1	30	7834	4293
Erode	4	0	2301	1452
Paintmask	1	0	2473	1410
Vivado HLS Video Library				
Threshold	1	0	1864	1207
AddWeighted	1	30	7481	5118
Erode	5	0	2493	1690
Paintmask	1	0	3306	1977

allel (i.e. one pixel per clock cycle). However, a different image format, like a planar one, where all of the samples in each channel are stored consecutively and the channels themselves are consecutive in memory as well (RRR...GGG...BBB...), would allow to process more elements in parallel (for instance, N elements belonging to the same channel per clock cycle). This is particularly true when we consider channel-independent filters. In this regard, In this regard, we demonstrate that FROST is able to design both interleaved and planar designs, thanks to the integration with its frontends. In particular, we implemented all the kernels in a channel interleaved manner, while, for the ones expressed in Tiramisu, we took advantage of Tiramisu features to also rearrange the input in a planar manner. Moreover, for each kernel we leveraged both FROST and the frontends scheduling co-language to evaluated different optimizations, while we chose a `streaming` dataflow architecture to implement such kernels, since the are implemented in the same way within the Video Library. Finally, we synthesized each considered design at 250MHz.

Halide benchmarks: For the Halide benchmarks, we chose the following kernels: `Threshold`, `AddWeighted`, `Erode`, and `PaintMask`. We designed each kernel in a channel interleaved manner, just like the HLS Video Library does. The resulting designs exploit both Halide and FROST scheduling commands to enable a parallel computation of the channels within each pixel (except the `Threshold` kernel, as it works on single-channel images). In Figure 5.7, we show the comparison, in terms of normalized execution time, between the FROST (with Halide) designs and the ones from the Video Library. FROST designs are able to match the perfor-

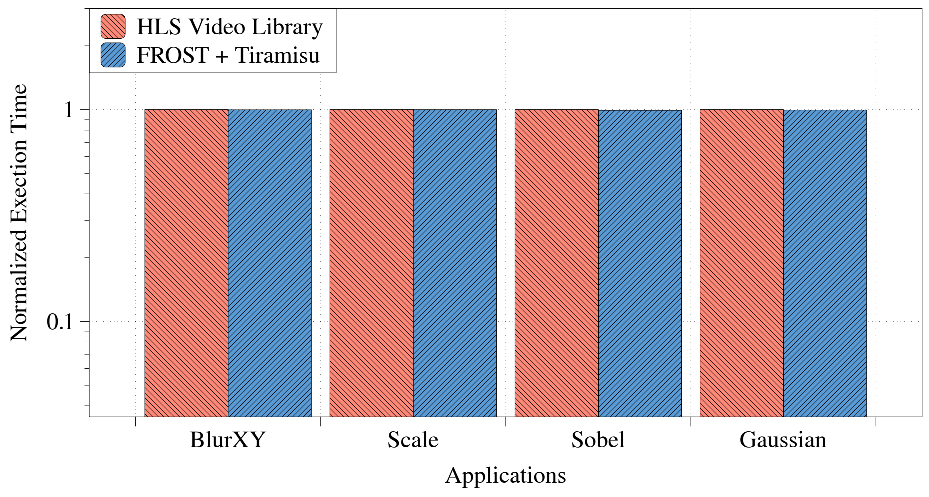

Figure 5.8: *Performance comparison between FROST with Tiramisu (interleaved designs) and Vivado HLS Video Library.*

mance of the HLS Video Library. Table 5.2 describes the resource usage of the considered kernels. We can notice that the resource usage of FROST designs is in line with the one of the Video Library.

Tiramisu benchmarks: For what concerns the Tiramisu benchmarks, we selected the following image processing kernels: `BlurXY`, `Scale`, `Sobel`, and `Gaussian`. For each kernel, we implemented both an interleaved and planar design. We relied on both FROST and Tiramisu scheduling commands to optimize the hardware designs. First of all, Tiramisu allowed us to prepare the computation for vectorization (applying loop splitting), and, when necessary, rearrange it in a planar manner. Then, FROST applied vectorization to the hardware designs and built a `streaming` dataflow architecture. On one hand, for the interleaved designs, we packed the 3 channels into a single variable, just like HLS Video Library does. On the other hand, the planar designs leveraged a higher level of parallelism, as we packed the input in chunks of 512-bit (i.e., 64 elements per chunk). This value represents the maximum bit-width of the memory ports of the DDR mounted on the target board. As we were using the full bit-width of the DDR ports, we mapped input/output buffers to different memory ports to fully exploit the available bandwidth (the VU9P board has 4 memory ports).

Figure 5.8 displays a comparison in terms of normalized execution time between FROST interleaved designs and the HLS Video Library designs,

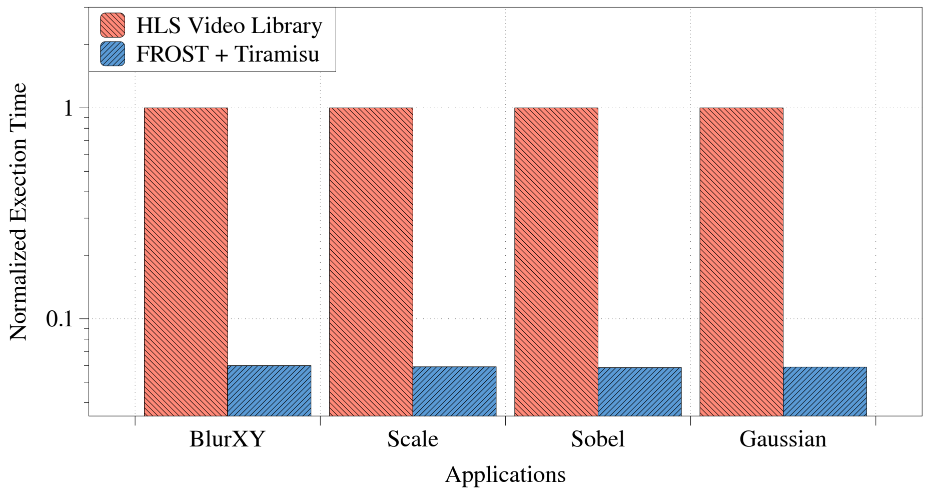

Figure 5.9: *Performance comparison between FROST with Tiramisu (planar designs) and Vivado HLS Video Library.*

while, in Figure 5.9, the comparison is between FROST planar designs and the Video Library. In Figure 5.8, we notice that FROST designs match the performance of HLS Video Library. In Figure 5.9, thanks to a higher level of parallelism, FROST designs significantly outperform the Video Library, reaching a speedup of 17X. Table 5.3 reports the resource usage of both FROST designs (interleaved and planar) and Vivado HLS Video Library designs. The interleaved designs have a resource usage similar to the Video Library, while the planar designs require a higher number of resources due to the higher level of parallelism.

5.5.3 Experimental Results: N-Body Simulation

In this Section, we compare an N-Body Simulation FPGA design produced by FROST, using Tiramisu as frontend, against state-of-the-art implementations on FPGA. The N-Body Simulation process describes the evolution of a system of forces composed of N bodies, which may represent celestial objects, molecules, and so on. The most accurate algorithm for N-Body simulation, the *All-Pairs* method, is particularly compute intensive, as it consists in a brute-force technique where all the pairwise interactions among the bodies are calculated. As a result, the computational complexity of such method is $O(N^2)$. We chose this application as benchmark because its high computational demand makes it a good candidate for hardware acceleration, as witnessed by different work in literature [167–172].

Table 5.3: *Resource usage of the Tiramisu benchmarks*

Application	*BRAM_18K* (4320)	*DSP48E* (6840)	*FF* (2364480)	*LUT* (1182240)
FROST (interleaved)				
BlurXY	4	0	2283	1342
Scale	1	15	4782	2696
Sobel	4	0	2214	1281
Gaussian	2	0	1835	1167
FROST (planar)				
BlurXY	30	0	10955	5098
Scale	15	320	68368	35979
Sobel	72	0	10381	5878
Gaussian	7	0	2589	1685
Vivado HLS Video Library				
BlurXY	5	12	2494	1853
Scale	1	15	4703	3704
Sobel	5	0	2532	1706
Gaussian	8	63	3316	2587

Moreover, differently from image processing kernels, this benchmark well fits a tiled architecture; hence, in this way, we can evaluate also this feature of FROST.

As stated before, we designed it in Tiramisu and optimized it for FPGA using FROST. Thanks to the co-scheduling language of FROST, we easily evaluated different solutions, and then implemented a design with multiple computational pipes. Table 5.4 reports the comparison between FROST design and the most relevant FPGA implementations of the N-Body All-Pairs method. We designed our accelerator with 96 computational pipes, just like the work in [172], which has the best $MPairs/s$ and performance/power ratio in the Table 5.4, and managed to synthesize it at 155.6 MHz. The design operates on 60,000 bodies at the time, and it is possible to deal with more bodies by invoking the design multiple times on different tiles of bodies.

As reported in the Table, our design reaches 13,069 MPairs/s and outperforms most of the works in literature, paying just about 3% loss in performance with respect to [172]. In terms of performance/power ratio, FROST implementation achieves 653.45 MPairs/s/W. As the target board is the same, the power consumption is in line with [172], thus the difference in performance/power ratio mainly depends on the performance difference.

Table 5.4: *Comparison between the proposed approaches and the All-Pairs N-Body implementations available in the literature.*

Platform	Cores/Pipelines	Performance [MPairs/s]	Performance/Power [MPairs/s/W]	Ref.
Vectis MAX3 Card	-	2,978	21.30	[168]
Xilinx VC707	32	2,327	116.36	[170]
VU9P	-	2,725	-	[169]
Arria 10	64	10,944	-	[171]
VU9P	96 ($3 \times 2 \times 16$)	13,441	672.06	[172]
VU9P	**96**	**13,069**	**653.45**	**FROST**

Table 5.5: *Resource usage of different N-Body designs*

Platform	BRAM_18K	DSP48E	FF	LUT	Ref.
Xilinx VC707	34%	48%	26%	85%	[170]
VU9P	84%	55%	27%	46%	[172]
VU9P	**19%**	**56%**	**18%**	**31%**	**FROST**

Finally, Table 5.5 contains the resource usage of both FROST and some state-of-the-art designs, as this information is not available in all the paper. We can notice that our design significantly uses the FPGA resources. In particular, Digital Signal Processors (DSPs) are the critical resource, reaching 56% of the available ones. Considering work targeting the same board, FROST design uses less resources with respect to the design in [172], which contains more logic to perform tiling in hardware.

5.6 Related Work

In the state of the art there are many and different tools whose purpose is to facilitate the hardware acceleration on FPGA of algorithms. HLS tools are examples of that. For instance, frameworks like Xilinx Vivado HLS [102], Intel HLS Compiler [173] and SDK for OpenCL [174] allow users to produce a RTL representation of a high level code, usually written in languages like C/C++ and OpenCL. HLS tools, like the aforementioned, support all computational domains, and feature a set of directives to guide the optimizations to apply to the resulting hardware design, as well as exhaustive reports describing the details of the design (e.g., resource usage, pipeline depths, etc.).

On the other hand, many frameworks and compilers focusing on specific contexts to generate efficient hardware implementations are emerging in literature. Darkroom [38] is a language and compiler embedded in Terra language [175] for image processing. Darkroom compiler takes as input a high level description of the application and translates it into line-buffered pipelines, expressed in Verilog HDL. Darkroom then synthesizes such pipelines for ASIC, FPGA, or CPU. The experimental evaluation of Darkroom reports gigapixel/sec performance for ASIC designs, while realtime 1080p/60 video processing for FPGAs ones. In [39], the authors present RIPL, a memory-efficient, declarative FPGA image processing DSL. At first, RIPL compiles the input programs into dataflow graphs, then it relies on an open source dataflow compiler [176] to generate the HDL. The authors evaluated RIPL against five benchmarks, and, without the need of synthesis directives, they showed a comparable memory usage with respect to the Vivado HLS Video Library [177]. The work in [40] presents an FPGA backend for the PolyMage DSL [178]. At first, the proposed backend enforces optimizations in terms of both data parallelism and memory bandwidth, then it leverages Vivado HLS to produce the FPGA design. In the experimental evaluation, the authors compared their backend against both Darkroom, and Vivado HLS Video Library. On average, this work reaches a $1.5\times$ speedup. ExaSlang 4 [41] is a DSL designed for the hardware acceleration on FPGA of numerical solvers based on the multigrid method. ExaSlang 4 takes advantage of Vivado HLS as backend to generate the hardware implementation of the input code. The presented approach outperformed a vectorized, single-threaded execution on an Intel i7 by a 3X factor. In [42], the authors describe an extension to Halide [32] to accelerate image processing kernels on Xilinx Zynq MPSoCs. To this end, the authors provided Halide with additional scheduling commands to control some crucial aspects of the resulting FPGA designs, like the depth of the FIFOs between kernels. Polyhedral compilers such as Rose [179] and PENCIL [33] use fully automatic techniques (such as the Pluto [180] scheduling algorithm) to parallelize and optimize computations and generate an OpenCL or HLS code that targets FPGA architectures.

With respect to the aforementioned work available in literature, FROST is designed as a common backend for multiple DSLs, instead of being specific to one DSL, and thus reduces the effort of developing a new FPGA backend. It is also designed to support data parallel algorithms implemented as loops and operating on dense arrays and tensors. One of the fundamental features of FROST is its high level scheduling co-language, which allows the user to specify exactly how the computation should be optimized and mapped to FPGA. Thanks to this feature, FROST is capable of being generic enough and to support general data parallel algorithms, while still reaching a high level of performance.

5.7 Final Remarks

DSLs are an efficient solution to both easily target heterogeneous architectures and increase productivity. Indeed, they permit users to quickly and easily develop portable and efficient designs for multiple architectures. However, in spite of the high support and efficiency when targeting CPUs and GPUs, DSLs still lack a concrete support for FPGAs. To this end, we designed FROST, a unified backend to efficiently hardware-accelerate DSLs on FPGAs. The design of FROST allows to support multiple DSLs as frontends, and leverages Xilinx SDAccel to generate the bitstream file. A crucial feature of FROST is a high-level scheduling co-language, which permits to optimize the resulting FPGA design according to the input application characteristics. In the experimental evaluation of FROST, we employed Halide and Tiramisu as frontends, and reached the same level of both performance and resource usage of a hand-tuned HLS library for image processing kernels (when we used the same level of parallelism), and outperformed it up to 17X thanks a combination of Tiramisu and FROST scheduling commands. On the other hand, FROST reached performance in line with hand-tuned state-of-the-art FPGA implementations of the N-Body Simulation All-Pairs method.

CHAPTER 6

Conclusions and Future Work

As described in the previous Chapters, the forthcoming end of Moore's Law [1] and Dennard's scaling [2] pushed researchers to investigate new solutions and innovate the computer architecture field. Over the last past decades, we first witnessed a significant shift in the trend of microprocessor design, from single core architectures to multi/many-cores. However, as the scaling of single core designs reached an end, multi-core scaling is following the same path, limited by Amdahl's Law [157]. Then, the trend shifted towards Heterogeneous System Architectures (HSAs). Heterogeneity seems a promising solution in order to increase performance and, at the same time, reduce the power consumption. Indeed, despite a higher management complexity, the combination of different processing units, namely CPUs, Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), and specialized Application Specific Integrated Circuits (ASICs), offers multiple solutions to efficiently execute the target workloads.

Different architectures provide different trade-offs in terms of performance, power/energy efficiency, and other metrics. Among the aforementioned architectures, FPGAs is the one offering the best trade-off. FPGAs are able to deliver higher performance than CPUs and similar to GPUs (higher for some workloads), while maintaining a relative low power profile (lower than CPUs and GPUs). In addition, even though FPGAs cannot reach the performance and power/energy efficiency of ASICs, reconfigurability feature makes them definitely more flexible and adaptable than ASICs. Therefore, FPGAs are for sure an effective solution in the context of HSAs, and, in general, as hardware accelerator. Nonetheless, the main limitation of FPGAs has historically been their programmability and their steep learning curve. Indeed, the development of efficient hardware designs for FPGA is a significantly complex task, and requires more time than similar CPUs and GPUs designs. Over the last years, there have been great improvements in FPGA tools, and the evolution of High-Level Synthesis (HLS) tools permitted to develop effective accelerators using languages like C, C++, and, recently, OpenCL. Although such innovations, the FPGA design still requires a high knowledge and expertise from the user, precluding or, at least, constraining, their usage to hardware designers.

Modern tools to program CPUs and GPUs offer a significant level of abstraction and productivity, and this is particularly true for domain specific tools, like Domain Specific Languages (DSLs) and Machine Learnings (MLs) frameworks. In such contexts, thanks to the restriction of the domain, languages like Halide [32] and/or tools like TensorFlow [35] permit users to fully focus on the computation itself, instead of its real implementation. In this way, the tools are completely in charge of implementing and optimizing the computation. The reduction of the domain also implies a reduction of the design space to explore, which allows domain specific compilers to converge to an optimal solution faster. In addition, domain specific tools usually provide backends not only for CPUs, but also for GPUs, which enable users to easily target multiple devices without a deep knowledge and expertise in designs for such architectures.

At the moment, the support for FPGAs within both DSLs and ML frameworks is still limited. In the former case, some DSLs are able to target FPGAs, but each DSL has its custom backend, and this reduce the possibility to easily create an FPGA backend for a specific DSL without starting from scratch. In the latter case, even though we will probably arrive to that point in the future, currently industrial ML frameworks do not officially and completely support FPGAs as backend. Given these motivations, the goal of this thesis is to bridge the gap in such contexts by providing tools able to abstract and facilitate FPGA design from domain specific tools. In this context, the contributions of this thesis are: a framework to automatize the design of FPGA accelerators from Convolutional Neural Networks (CNNs), highly integrated with ML frameworks like TensorFlow and Caffe [36]; a common backend to effectively target FPGAs from DSLs, which also offers a high-level scheduling co-language to express the optimize and customize the resulting design.

6.1 Limitations and Future Work

Although efficient, the proposed tools still have some limitations and there is room for further improvements. In this Section we describe the possible future work of both the CNNs framework and FROST.

6.1.1 CNN framework

Currently, the proposed CNN framework supports a limited number of possible CNN layers, consequently reducing the CNN models the users can implement. In terms of resources, the hardware designs generated by the framework are massively parallel and rely on 32-bit floating-point operations. This implies a significant usage of FPGA resources like programmable logic and Digital Signal Processors (DSPs), thus limiting the possibility of implementing deep networks.

Thanks to its modularity and the customizability of its hardware libraries, the proposed framework is flexible and scalable enough to permit the add new features able to surpass the aforementioned limitations. Indeed, from a framework prospective, the introduction of additional layers, like activations layer for Rectified Linear Unit (ReLU) or hyperbolic tangent, as well as other features is straightforward. With respect to the hardware designs, an efficient solution to reduce DSPs usage is the adoption of low/fixed-precision data types, leveraging the FPGA capability to implement operations with a custom bitwidth. Moreover, we plan to include in the hardware libraries additional parameters to tune the level of parallelism of each layer. This would allow to choose the right trade-off between resource usage and performance.

6.1.2 FROST

Currently, we evaluated FROST with Halide, an image processing DSL, and Tiramisu, an optimization framework for high performance system. Although Tiramisu is more general than Halide and permits to describe other types of computations in addition to image processing ones, we still have to fully evaluate FROST with DSLs targeting other domains, like linear algebra. This does not mean that FROST cannot currently handle such computations; indeed, FROST scheduling co-language already permits to design a tiled architecture, which may be more suitable for linear algebra computations with respect to a streaming one. However, only after integrating one or more DSLs of this kind, it would be possible to better understand if additional scheduling commands oriented to such domain are necessary.

Given this motivation, as future work, first of all, we plan to add support for additional DSLs that are not currently supported. This will require to implement an ad-hoc translator from such DSLs to FROST Intermediate Representation (IR), as we did for Halide and Tiramisu. Then, as said before, we intend to introduce other high-level scheduling commands, like one for tree-reduction in case of accumulations, to further improve the productivity and efficiency of FROST. We would also like to better integrate FROST with Xilinx SDAccel. This would allow to invoke SDAccel directly from FROST and enable users to easily evaluate the performance of a design, optimize it according to the HLS phase results, and, once satisfied, start the compilation process by means of the script already provided by FROST. In addition to that, we would like to provide FROST with an auto-tuner. With respect to that, we could either implement an auto-tuner from scratch or rely on state-of-the-art tools like OpenTuner [181], similarly to what already happens with Halide. Finally, we plan to release FROST as open source.

6.2 Final Remarks

As said before, FPGA are gaining more and more attention as hardware accelerators. As matter of fact, nowadays many companies offer cloud instances powered by FPGAs. Some examples are: Alibaba Cloud F2 instances [182], Amazon EC2 F1 instances [183], Huawei FPGA Accelerated Cloud Server [184], Baidu FPGA Cloud Server [185], Nimbix FPGA instances [186], Tencent FPGA Cloud Server [187], and so on. In addition to that, such services also provide some libraries of ready-to-use IPs to accelerate certain types of computations, like ML ones. This demonstrates the increasing interest of companies towards FPGAs as an alternative to CPUs and GPUs, and their effort to reduce the complexity of FPGA design.

On the other hand, the restriction of the application domain permits to both further boost performance, and design tools capable of abstracting the complexity at algorithmic level and tailoring compiler optimizations to the target context. As a consequence, domain specific tools and architectures are one of the current and future trends in computer science. Indeed, as also stated by Prof. Hennessy and Prof. Patterson during their Turing Lecture [188], Domain Specific Architectures (DSAs) represent the next step in the compute architecture scenario towards major improvements in performance-cost-energy. A significant example of this trend is the Google Tensor Processing Unit (TPU) [134]. Also in this context, FPGAs may become an effective solution, given the possibility to evaluate different architectural choices and easily upgrade and improve current designs, without the need of producing a new device, as it happens with ASICs.

This work is in-between of these two scenarios. On one hand, its purpose is to ease and abstract the usage of FPGAs to make them more appealing to non-hardware designers. On the other, it focuses on domain specific tools, like DSLs and ML frameworks, and leverages the features and peculiarities of such domains to generate efficient hardware designs. In conclusion, the final goal of this thesis is to contribute to the research in this field and be one of the building blocks for the future generation of domain specific solutions.

Bibliography

[1] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, pages 114–117, Apr 1965.

[2] R.H. Dennard, F.H. Gaensslen, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted mosfet’s with very small physical dimensions. 9(5):256–268, Oct 1974.

[3] Intel. Microprocessor Quick Reference Guide. `http://www.intel.com/pressroom/kits/quickreffam.htm`.

[4] Intel. The Story of the Intel 4004. `http://www.intel.com/content/www/us/en/history/museum-story-of-intel-4004.html`.

[5] Intel. Intel Xeon Phi Processor 7290F. `https://ark.intel.com/products/95831/Intel-Xeon-Phi-Processor-7290F-16GB-1-50-GHz-72-core-`.

[6] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, May 2011.

[7] Akihiko Miyoshi, Charles Lefurgy, Eric Van Hensbergen, Ram Rajamony, and Raj Rajkumar. Critical power slope: Understanding the runtime effects of frequency scaling. In *Proceedings of the 16th International Conference on Supercomputing*, ICS ’02, pages 35–44, New York, NY, USA, 2002. ACM.

[8] D Pham, S Asano, M Bolliger, MN Day, HP Hofstee, C Johns, J Kahle, A Kameyama, J Keaty, Y Masubuchi, et al. The design and implementation of a first-generation cell processor-a multi-core soc. In *Integrated Circuit Design and Technology, 2005. ICICDT 2005. 2005 International Conference on*, pages 49–52. IEEE, 2005.

[9] Jeff Parkhurst, John Darringer, and Bill Grundmann. From single core to multi-core: Preparing for a new exponential. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design*, ICCAD ’06, pages 67–72, New York, NY, USA, 2006. ACM.

[10] Mark D. Hill and Michael R. Marty. Amdahl’s law in the multicore era. *Computer*, 41(7):33–38, 2008.

[11] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, May 2005.

[12] Intel. 6th Generation Intel Core i5 Processors. `http://www.intel.com/content/www/us/en/processors/core/core-i5-processor.html`.

[13] Intel. 6th Generation Intel Core i7 Processors. http://www.intel.com/content/www/us/en/processors/core/core-i7-processor.html.

[14] AMD. AMD A-Series APU Processors. http://www.amd.com/en-us/products/processors/desktop/a-series-apu.

[15] Intel. Intel Xeon Processor E7 Family. http://www.intel.com/content/www/us/en/processors/xeon/xeon-processor-e7-family.html.

[16] IBM. IBM Power Systems. http://www-03.ibm.com/systems/power/index.html.

[17] Oracle. Oracle SPARC Systems. https://www.oracle.com/servers/sparc/index.html.

[18] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[19] David Patterson. 50 years of computer architecture: From the mainframe cpu to the domain-specific tpu and the open risc-v instruction set. In *Solid-State Circuits Conference-(ISSCC), 2018 IEEE International*, pages 27–31. IEEE, 2018.

[20] HSA Fundation. http://www.hsafoundation.com.

[21] Standard Performance Evalutation Corporation. SPEC Benchmarks. https://www.spec.org/benchmarks.html.

[22] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017.

[23] TOP500 List - June 2018. https://www.top500.org/lists/2018/06/.

[24] Oak Ridge National Laboratory. Summit. https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/.

[25] Wikichip. Matrix-2000. https://en.wikichip.org/wiki/nudt/matrix-2000.

[26] Green500 List - June 2018. https://www.top500.org/green500/lists/2018/06/.

[27] Intel, Inc. Intel Xeon Processor D-1571. https://ark.intel.com/products/93355/Intel-Xeon-Processor-D-1571-24M-Cache-1-30-GHz-.

[28] Wikichip. PEZY-SC2 Many Core Processor. https://en.wikichip.org/wiki/pezy/pezy-scx/pezy-sc2.

[29] Khronos Group Inc. OpenCL. https://www.khronos.org/opencl/.

[30] NVIDIA. CUDA. https://developer.nvidia.com/about-cuda.

[31] Open MPI Project. https://www.open-mpi.org.

[32] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[33] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F Donaldson, Jeroen Ketema, et al. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*, pages 138–149. IEEE, 2015.

[34] NVIDIA, Inc. Machine Learning. `https://www.nvidia.com/object/machine-learning.html`.

[35] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems. `http://tensorflow.org/`, 2015.

[36] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[37] Torch Framework. `http://torch.ch`.

[38] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom: compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.*, 33(4):144–1, 2014.

[39] Robert Stewart, Greg Michaelson, Deepayan Bhowmik, Paulo Garcia, and Andy Wallace. A dataflow IR for memory efficient RIPL compilation to FPGAs. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 174–188. Springer, 2016.

[40] Nitin Chugh, Vinay Vasista, Suresh Purini, and Uday Bondhugula. A DSL compiler for accelerating image processing pipelines on FPGAs. In *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*, pages 327–338. IEEE, 2016.

[41] Christian Schmitt, Moritz Schmid, Frank Hannig, Jürgen Teich, Sebastian Kuckuk, and Harald Köstler. Generation of multigrid-based numerical solvers for FPGA accelerators. In *Proceedings of the 2nd International Workshop on High-Performance Stencil Computations (HiStencils)*, pages 9–15, 2015.

[42] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming heterogeneous systems from an image processing dsl. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):26, 2017.

[43] E. Del Sozzo, A. Solazzo, A. Miele, and M. D. Santambrogio. On the automation of high level synthesis of convolutional neural networks. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 217–224, May 2016.

[44] A. Solazzo, E. D. Sozzo, I. De Rose, M. D. Silvestri, G. C. Durelli, and M. D. Santambrogio. Hardware design automation of convolutional neural networks. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 224–229, July 2016.

[45] E. Del Sozzo, R. Baghdadi, S. Amarasinghe, and M. D. Santambrogio. A common backend for hardware acceleration on fpga. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 427–430, Nov 2017.

[46] E. Del Sozzo, R. Baghdadi, S. Amarasinghe, and M. D. Santambrogio. A unified backend for targeting fpgas from dsls. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 1–8, July 2018.

[47] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. P. Amarasinghe. TIRAMISU: A Polyhedral Compiler for Expressing Fast and Portable Code. In *2019 ACM International Symposium on Code Generation and Optimization (CGO)*, February 2019.

[48] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. 31(4):6–15, July 2011.

[49] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Power limitations and dark silicon challenge the future of multicore. *ACM Transactions on Computer Systems*, 30(3):11:1–11:27, August 2012.

[50] P. Kogge and J. Shalf. Exascale computing trends: Adjusting to the "new normal" for computer architecture. *Computing in Science & Engineering*, 15(6):16–26, Nov 2013.

[51] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[52] AMD. High-performance computing. `https://www.amd.com/en-us/products/graphics/workstation/firepro-remote-graphics/gpu-compute`.

[53] Scott Hauck and Andre DeHon, editors. *Reconfigurable computing: the theory and practice of FPGA-based computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, first edition, 2010.

[54] Xilinx Inc. 7 series fpgas configurable logic block. `https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf`.

[55] R.A. Walker and D.E. Thomas. A model of design representation and synthesis. In *22nd (DAC '85)*, pages 453–459, June 1985.

[56] D.D. Gajski and R.H. Kuhn. Guest editors' introduction: New vlsi tools. 16(12):11–14, Dec 1983.

[57] Katherine Compton and Scott Hauck. Reconfigurable computing: A survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, June 2002.

[58] Xilinx Inc. Vivado Design Suite. `http://www.xilinx.com/products/design-tools/vivado.html`.

[59] Synopsys. RTL Synhtesis and Test. `https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test.html`.

[60] Mentor Graphics. Design Creation. `https://www.mentor.com/products/fpga/hdl_design/`.

[61] Shimpei Sato and Kenji Kise. Archhdl: A novel hardware rtl design environment in c++. In *Applied Reconfigurable Computing*, pages 53–64. Springer, 2015.

[62] Rishiyur Nikhil. Bluespec: A general-purpose approach to high-level synthesis based on parallel atomic transactions. In *High-Level Synthesis*, pages 129–146. Springer, 2008.

[63] Rishiyur Nikhil. Tutorial bluespec systemverilog: Efficient, correct rtl from high-level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, pages 69–70, 2004.

[64] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. Chisel: Constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, June 2012.

[65] Ofer Shacham, Sameh Galal, Sabarish Sankaranarayanan, Megan Wachs, John Brunhaver, Artem Vassiliev, Mark Horowitz, Andrew Danowitz, Wajahat Qadeer, and Stephen Richardson. Avoiding game over: Bringing design to the next level. In *DAC Design Automation Conference*, pages 623–629. IEEE, 2012.

[66] Yanbing Li and Miriam Leeser. Hml, a novel hardware description language and its translation to vhdl. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(1):1–8, 2000.

[67] Peter Bellows and Brad Hutchings. Jhdl - an hdl for reconfigurable systems. In *IEEE Symposium on FPGAs for Custom Computing Machines*. IEEE, 1998.

[68] Maxeler Inc. Multiscale dataflow programming. `https://www.maxeler.com/products/software/maxcompiler/`.

[69] Jan Decaluwe. Myhdl: a python-based hardware description language. `https://www.linuxjournal.com/article/7542`.

[70] Ali Mashtizadeh. Phdl: A python hardware design framework. `https://dspace.mit.edu/handle/1721.1/41543`.

[71] Derek Lockhart, Gary Zibrat, and Christopher Batten. Pymtl: A unified framework for vertically integrated computer architecture research. In *47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 280–292. IEEE, 2014.

[72] Verilator. `https://www.veripool.org/projects/verilator/`.

[73] Shinya Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *Applied Reconfigurable Computing*, pages 451–460. Springer, 2015.

[74] Veriloggen. `https://github.com/PyHDI/veriloggen`.

[75] Christian Pilato and Fabrizio Ferrandi. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–4. IEEE, 2013.

[76] Bambu: A free framework for the high-level synthesis of complex applications. `https://panda.dei.polimi.it/?page_id=31`.

[77] Thomas Bollaert. Catapult synthesis: a practical introduction to interactive C synthesis. In *High-Level Synthesis*, pages 29–52. Springer, 2008.

[78] Mentor Graphics. Catapult High-Level Synthesis. `https://www.mentor.com/hls-lp/catapult-high-level-synthesis/`.

[79] Impulse Accelerated Technologies. CoDeveloper Help Contents. `http://www.impulsec.com/ReleaseFiles/Help/iAppMan.pdf`.

[80] Impulse Accelerated Technologies. Impulse C User Guide. `http://www.impulsec.com/ReleaseFiles/Help/ImpulseCUserGuide.pdf`.

[81] NEC. CyberWorkBench: High Level Synthesis from C/C++/SystemC to ASIC/FPGA. `https://www.nec.com/en/global/prod/cwb/index.html`.

[82] Mentor Graphics. DK Design Suite. `https://www.mentor.com/products/fpga/handel-c/dk-design-suite/`.

[83] Razvan Nane, Vlad-Mihai Sima, Bryan Olivier, Roel Meeuws, Yana Yankova, and Koen Bertels. Dwarv 2.0: A cosy-based c-to-vhdl hardware compiler. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 619–622. IEEE, 2012.

[84] ACE. CoSy compiler development system. `http://www.ace.nl/compiler/cosy`.

[85] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016.

[86] YXI. eXCite. `http://www.yxi.com/products.php`.

[87] Philippe Coussy, Cyrille Chavet, Pierre Bomel, Dominique Heller, Eric Senn, and Eric Martin. Gaut: A high-level synthesis tool for dsp applications. In *High-Level Synthesis*, pages 147–169. Springer, 2008.

[88] Universitè Bretagne Sud. GAUT – High-Level Synthesis Tool from C to RTL. http://hls-labsticc.univ-ubs.fr/.

[89] Philippe Coussy, Ghizlane Lhairech-Lebreton, Dominique Heller, and Eric Martin. Gaut – a free and open source high-level synthesis tool. In *IEEE DATE*, 2010.

[90] Intel. Intel HLS Compiler - Overview. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

[91] Intel. Intel HLS Compiler - Reference Manual. https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/hls/mnl-hls-reference.pdf.

[92] LegUp Computing. High-Level Synthesis For Any FPGA. https://www.legupcomputing.com/.

[93] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM, 2011.

[94] ECE Department, University of Toronto. LegUp High-Level Synthesis. http://hls-labsticc.univ-ubs.fr/.

[95] Jason Villarreal, Adrian Park, Walid Najjar, and Robert Halstead. Designing modular hardware accelerators in c with roccc 2.0. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 127–134. IEEE, 2010.

[96] University of California, Riverside. ROCCC 2.0. http://roccc.cs.ucr.edu/.

[97] Muhsen Owaida, Nikolaos Bellas, Konstantis Daloukas, and Christos D Antonopoulos. Synthesis of platform architectures from opencl programs. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 186–193. IEEE, 2011.

[98] Cadence. Impulse C User Guide. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html.

[99] David Pursley and Tung-Hua Yeh. High-level low-power system design optimization. In *VLSI Design, Automation and Test (VLSI-DAT), 2017 International Symposium on*, pages 1–4. IEEE, 2017.

[100] Synopsys. Synphony C Compiler. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/synphony-c-compiler.html.

[101] Berkeley Design Technology Inc. BDTI High-Level Synthesis Tool Certification Program Results. https://www.bdti.com/Resources/BenchmarkResults/HLSTCP.

[102] Xilinx Inc. Vivado HLS. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[103] Intel. Intel FPGA SDK for OpenCL. https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html.

[104] Xilinx Inc. SDAccel Development Environment. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[105] Xilinx Inc. SDSoC Development Environment. https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html.

[106] IBM, Inc. 2.5 quintillion bytes of data created every day. How does CPG & Retail manage it? https://www.ibm.com/blogs/insights-on-business/consumer-products/2-5-quintillion-bytes-of-data-created-every-day-how-does-cpg-retail-manage-it/aaa.

[107] Domo, Inc. Data Never Sleeps 6.0. https://www.domo.com/learn/data-never-sleeps-6.

[108] Kaggle, Inc. 2017 The State of Data Science & Machine Learning. https://www.kaggle.com/surveys/2017, 2017.

[109] Amazon.com, Inc. AWS Rekognition. https://aws.amazon.com/rekognition/.

[110] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24. IEEE, 2014.

[111] Apple Inc. Siri. http://www.apple.com/ios/siri/.

[112] Google Inc. Google Photos. https://photos.google.com/.

[113] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 05 2015.

[114] Riccardo Petrolo, Valeria Loscrí, and Nathalie Mitton. Towards a smart city based on cloud of things. In *Proceedings of the 2014 ACM International Workshop on Wireless and Mobile Technologies for Smart Cities*, WiMobCity '14, pages 61–66, New York, NY, USA, 2014. ACM.

[115] P. A. Laplante and N. Laplante. The Internet of Things in Healthcare: Potential Applications and Challenges. *IT Professional*, 18(3):2–4, May 2016.

[116] Elias De Coninck, Tim Verbelen, Bert Vankeirsbilck, Steven Bohez, Sam Leroux, and Pieter Simoens. Dianne: Distributed artificial neural networks for the internet of things. In *Proceedings of the 2Nd Workshop on Middleware for Context-Aware Applications in the IoT*, M4IoT 2015, pages 19–24, New York, NY, USA, 2015. ACM.

[117] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Comput. Netw.*, 54(15):2787–2805, October 2010.

[118] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, page 2012.

[119] C. Farabet, C. Couprie, L. Najman, and Y. LeCun. Learning hierarchical features for scene labeling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1915–1929, Aug 2013.

[120] Jonathan Tompson, Arjun Jain, Yann LeCun, and Christoph Bregler. Joint training of a convolutional network and a graphical model for human pose estimation. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'14, pages 1799–1807, Cambridge, MA, USA, 2014. MIT Press.

[121] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[122] T. Mikolov, A. Deoras, D. Povey, L. Burget, and J. Cernocky. Strategies for training large scale neural network language models. In *2011 IEEE Workshop on Automatic Speech Recognition Understanding*, pages 196–201, Dec 2011.

[123] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, Nov 2012.

[124] T. N. Sainath, A. Mohamed, B. Kingsbury, and B. Ramabhadran. Deep convolutional neural networks for lvcsr. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8614–8618, May 2013.

[125] Junshui Ma, Robert P. Sheridan, Andy Liaw, George E. Dahl, and Vladimir Svetnik. Deep neural nets as a method for quantitative structure-activity relationships. *Journal of Chemical Information and Modeling*, 55(2):263–274, 2015. PMID: 25635324.

[126] Moritz Helmstaedter, Kevin L. Briggman, Srinivas C. Turaga, Viren Jain, H. Sebastian Seung, and Winfried Denk. Connectomic reconstruction of the inner plexiform layer in the mouse retina. *Nature*, 500:168–174, 2013.

[127] Michael K. K. Leung, Hui Yuan Xiong, Leo J. Lee, and Brendan J. Frey. Deep learning of the tissue-regulated splicing code. In *Bioinformatics*, 2014.

[128] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *J. Mach. Learn. Res.*, 12:2493–2537, November 2011.

[129] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proc. of the IEEE*, 86(11):2278–2324, 1998.

[130] Nicholas D. Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, and Fahim Kawsar. An early resource characterization of deep learning on wearables, smartphones and internet-of-things devices. In *Proceedings of the 2015 International Workshop on Internet of Things Towards Applications*, IoT-App '15, pages 7–12, New York, NY, USA, 2015. ACM.

[131] M. Peemen, A.A.A. Setio, B. Mesman, and H. Corporaal. Memory-centric accelerator design for convolutional neural networks. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pages 13–19, 2013.

[132] Murugan Sankaradas, Venkata Jakkula, Srihari Cadambi, Srimat Chakradhar, Igor Durdanovic, Eric Cosatto, and Hans Peter Graf. A massively parallel coprocessor for convolutional neural networks. In *Proc. of IEEE Int. Conf. on Application-specific Systems, Architectures and Processors ASAP)*, pages 53–60, 2009.

[133] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proc. of ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 161–170, 2015.

[134] Google Tensor Processing Unit. `https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html`.

[135] Protocol Buffers. `https://developers.google.com/protocol-buffers/`.

[136] Y LeCun, B Boser, JS Denker, D Henderson, RE Howard, W Hubbard, and LD Jackel. Handwritten digit recognition with a back-propagation network. In *Proceedings of the 2nd International Conference on Neural Information Processing Systems*, pages 396–404. MIT Press, 1989.

[137] MNIST handwritten digits dataset. `http://yann.lecun.com/exdb/mnist/`.

[138] Frank Rosenblatt. The perceptron a perceiving and recognizing automaton. Technical report, tech. rep., Technical Report 85-460-1, Cornell Aeronautical Laboratory, 1957. 2, 1957.

[139] Michael Nielsen. *Neural Networks and Deep Learning*.

[140] Anil Kumar Goswami, Shalini Gakhar, and Harneet Kaur. Automatic object recognition from satellite images using artificial neural network. *International Journal of Computer Applications*, 95(10), 2014.

[141] David H Hubel and Torsten N Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195(1):215–243, 1968.

[142] Google, Inc. Protocol Buffers Language Guide. `https://developers.google.com/protocol-buffers/docs/proto`, 2018.

[143] Xilinx Inc. Zybo Zynq-7000 Development Board. `http://www.xilinx.com/products/boards-and-kits/1-4azfte.html`.

[144] Zedboard. `http://zedboard.org/product/zedboard`.

[145] Xilinx Inc. Virtex-7 FPGAs. `https://www.xilinx.com/products/silicon-devices/fpga/virtex-7.html`.

[146] Voltcraft. `http://www.voltcraft.com`.

[147] D. Strigl, K. Kofler, and S. Podlipnig. Performance and Scalability of GPU-Based Convolutional Neural Networks. In *Proc. of Euromicro Int. Conf. on Parallel, Distributed and Network-Based Processing (PDP)*, pages 317–324, 2010.

[148] Dan C Ciresan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *Proc. of Int. Joint Conf. on Artificial Intelligence*, volume 22, page 1237, 2011.

[149] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello. Hardware accelerated convolutional neural networks for synthetic vision systems. In *Proc. of IEEE Int. Symp. on Circuits and Systems (ISCAS)*, pages 257–260, 2010.

[150] M. Peemen, A.A.A. Setio, B. Mesman, and H. Corporaal. Memory-centric accelerator design for convolutional neural networks. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pages 13–19, Oct 2013.

[151] Maurice Peemen, Bart Mesman, and Henk Corporaal. Inter-tile Reuse Optimization Applied to Bandwidth Constrained Embedded Accelerators. In *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 169–174, 2015.

[152] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun. CNP: An FPGA-based processor for Convolutional Networks. In *Proc. of Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 32–37, 2009.

[153] Yongming Shen, Michael Ferdman, and Peter Milder. Overcoming Resource Underutilization in Spatial CNN Accelerators. In *Proc. of Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2016.

[154] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In *Proc.of ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, pages 26–35, 2016.

[155] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[156] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun. CNP: An FPGA-based processor for Convolutional Networks. In *Proc. of Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 32–37, 2009.

[157] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[158] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, December 1992.

[159] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*, 2012.

[160] James Bergstra, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins, Ian Goodfellow, Arnaud Bergeron, Yoshua Bengio, and Pack Kaelbling. Theano: Deep learning on gpus with python, 2011.

[161] Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. Distributed halide. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 5. ACM, 2016.

[162] Facebook Inc. Annuncing Tensor Comprehensions. `https://research.fb.com/announcing-tensor-comprehensions/`.

[163] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018.

[164] Mary W Hall, Saman P Amarasinghe, Brian R Murphy, Shih-Wei Liao, and Monica S Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*, pages 49–49. IEEE, 1995.

[165] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael FP O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. *ACM Sigplan Notices*, 44(6):177–187, 2009.

[166] Xilinx Inc. AMBA AXI4 Interface Protocol. `https://www.xilinx.com/products/intellectual-property/axi.html`.

[167] Junichiro Makino and Hiroshi Daisaka. Grape-8–an accelerator for gravitational n-body simulation with 20.5 gflops/w performance. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–10. IEEE, 2012.

[168] Maxeler Technologies. N-body, 2015.

[169] Lorenzo Di Tucci, Marco Rabozzi, Luca Stornaiuolo, and Marco D Santambrogio. The role of cad frameworks in heterogeneous fpga-based cloud systems. In *Computer Design (ICCD), 2017 IEEE International Conference on*, pages 423–426. IEEE, 2017.

[170] Emanuele Del Sozzo, Lorenzo Di Tucci, and Marco Domenico Santambrogio. A highly scalable and efficient parallel design of n-body simulation on fpga. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 241–246, May 2017.

[171] Kentaro Sano, Shin Abiko, and Tomohiro Ueno. Fpga-based stream computing for high-performance n-body simulation using floating-point dsp blocks. In *Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, page 16. ACM, 2017.

[172] E. Del Sozzo, M. Rabozzi, L. Di Tucci, D. Sciuto, and M. D. Santambrogio. A scalable fpga design for cloud n-body simulation. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 1–8, July 2018.

[173] Intel Inc. Intel HLS Compiler. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01274-intel-hls-compiler-fast-design-coding-and-hardware.pdf.

[174] Intel Inc. Intel FPGA SDK for OpenCL. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_getting_started.pdf.

[175] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: a multi-stage language for high-performance computing. In *ACM SIGPLAN Notices*, volume 48, pages 105–116. ACM, 2013.

[176] Endri Bezati. *High-level synthesis of dataflow programs for heterogeneous platforms*. PhD thesis, EPFL, 2015.

[177] Xilinx Inc. Xilinx Vivado HLS Video Library. http://www.wiki.xilinx.com/HLS+Video+Library.

[178] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. Polymage: Automatic optimization for image processing pipelines. *SIGARCH Comput. Archit. News*, 43(1):429–443, March 2015.

[179] Louis-Noel Pouchet, Peng Zhang, Ponnuswamy Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 29–38. ACM, 2013.

[180] Uday Bondhugula, A Hartono, J Ramanujam, and P Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer, 2008.

[181] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O’Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT ’14, pages 303–316, New York, NY, USA, 2014. ACM.

[182] Alibaba. Alibaba Cloud. https://www.alibabacloud.com/about?spm=a3c0i.7911826.675768.dnavwhya2.2d00968evRW8ZQ.

[183] Amazon. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/.

[184] Huawei. FPGA Accelerated Cloud Server. https://www.huaweicloud.com/product/fcs.html.

[185] Baidu. Baidu FPGA Compute Cloud. https://cloud.baidu.com/product/fpga.html.

[186] Nimbix. Xilinx Alveo Accelerator Cards. https://www.nimbix.net/alveo/.

[187] Tencent. FPGA Compute Cloud. https://cloud.tencent.com/product/fpga?lang=en.

[188] Hennessy, John and Patterson, David. Turing Lecture. http://iscaconf.org/isca2018/turing_lecture.html, 2018.