POLITECNICO
MILANO 1863

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

ON THE ROLE OF RECONFIGURABLE SYSTEMS IN DOMAIN-SPECIFIC COMPUTING

Doctoral Dissertation of:
Davide Conficconi

Supervisor:
Prof. Marco D. Santambrogio

Tutor:
Prof. Cristina Silvano

The Chair of the Doctoral Program:
Prof. Luigi Piroddi

2022 – XXXIV Cycle

I would like to take this space to say a huge thanks to many people I had the luck to meet. I am grateful for everyone who met during this incredible journey, that had a role, either for good or bad. I do believe that people make the difference, at least for me, and for this dissertation, they do.

Thanks to all the coauthors of the research manuscript that this dissertation refers to. Collaborating with all of you has been an honor for me and taught me something in all these situations. Thanks to: Eleonora, Prof. Santambrogio, Prof. Sciuto, Emanuele, Alberto Z., Mirko, Yaman, Lahiru, Thomas, Prof. Själander, Alessandro, Alberto S., Filippo, Daniele Par., Prof. Pilato, Enrico, Giuseppe, Carlo, Daniele Pal..

Thanks to Michaela Blott for hosting me during my research period at Xilinx Ireland, the other interns Lois, Alex, Chris, Lisa, and the whole research team Yaman, Alessandro, Giulio, Nick, Ken, Lucian, as well as Lin Ya, Cathal, Peter, Giuseppe and Baris's and David's team! I also want to thank the guys of the X-basket who helped me during those hard times: Wojciech, Georgis, Aitor, Alex, Guillermo, Ines, Pedro, and all the others! I also want to thank Mark Lantz for currently hosting me at IBM Zurich and all the cloudFPGA team: Dionysios, Beat, Francois, Burkhard, Florian, Christoph, and the Hobby club Yoga, with a particular mention to Stephan. Although the pandemic, I have been very lucky. These research stays had been an incredible opportunity for growth. Thanks.

Thanks to my advisor Marco who believed (and hopefully still believes) in me and gave me this great chance to explore and experiment across several different things: from research to teaching. Thanks also for pushing a place like the NECSTLab, which is full of incredible and heterogeneous people. Thanks to all the people of the NECSTLab with whom I shared a piece or the whole PhD journey for their spirit, diversity and points of view, laugh, and hard work. Thanks to all that passed there for a while or are still here. Thanks to the security group: John, Ste, Mario, Stefano, Shamano, Armando and all the others. Thanks to whole PhD++ team (past and present): Alberto S., Emanuele, Giuseppe, Rabboz, Lorenzo, Sara, Rolando, Luca, Arna, Carlotta, Parra, Ele, Andre, Ziner, Gw, Mirko, Leti, Silvia, Filippo, Francesco P., Francesco S., Isabella, Ilaria Mos., Ilaria Mor., e Fulvia. A special thanks to Emanuele, friend, colleague, and unofficial coadivsor of my PhD. Thanks for all the effort and for your time. Thanks to Filippo, now PhD student, but friend and my first coadvising master student. I hope I have been a good coadvisor for the future tuning award.

Thanks to my family, who had never stopped believing in me, although I have never had been a good student. Thanks for the infinite opportunities that my parents, Cristina and Antonio, gave me. Thanks to my sisters and brother Elisa, Chiara, Myriam, Michele that one way or another are always there. Thanks to Pietro, Matteo, and the newcomers Arianna and Giorgio (?).

Thanks to other friends who did not contribute directly to research, but your presence had a terrific indirect contribution. Thanks to Elisabetta, Paolo, Pier, Elia, and Lara for the walks, the laughs, and the chats. Thanks to the Owowiwowa group for the infinite source of laugh and ideas: Angelo, Chicco, Dani, Edo, Jack, Ilyas, Ste, Yanni, Ale, Paul. Thanks to Alessandro for being present, even being in a different country. Thanks to Totta, Andre, and Pippo, for the never-ending adventures.

Last but not least, thanks to Eleonora for sharing her emotions, ideas, and time with me. It is an incredible journey, and I am lucky to be sharing it with you.

Thanks also to all of you that I did not directly mention. I hope you can apologize to me, but can genuinely believe I appreciate we met and spent some time.

This dissertation has been possible for all of you.

Thank you.

ヨホホホ　ヨホホホ♪

Abstract

COMPUTER architectures field is facing technological and architectural obstacles that are limiting the general-purpose processor scaling in the delivered performance at a reasonable energy cost. Therefore, computer architects have to follow novel paths to harvest more energy-efficient computations from the currently available technology, for instance, by employing *domain specialized* solutions for a given scenario. Indeed, domain-specialized architectures can deliver extremely high performance at a relatively low energy profile and even more whenever combined with high-level abstractions for designing and programming it. Domain-Specific Architectures (DSAs) generally are the prominent exponent for domain specialization. Moreover, DSAs are programmable software architectures designed for few tasks, with the minimal amount of advanced CPU-based microarchitectural techniques, and to be efficiently implemented as Application-Specific Integrated Circuits (ASICs), or part of System on Chip (SoC). However, developing custom silicon devices is a time-consuming and costly process that is not always compatible with the time-to-market and fast evolution of the applications. Thus, adaptable computing platforms represent the most viable alternative for these scenarios. Field-Programmable Gate Arrays (FPGAs) are the candidate platforms for their *on-field* reconfigurable heterogeneous fabric. On top of the reconfigurability, FPGAs can implement large spatial computing designs and are publicly available on cloud computing platforms.

Therefore, this dissertation focus on *Domain-Specific Reconfigurable Architectures* (DSRAs): domain-specialized architectures with *adaptable*

datapaths implemented on FPGAs. The design of such architectures demands a clear view of system-level trends and FPGAs' abstraction layers. On top of that, advanced design methodologies enable domain-tailored energy-efficient architectures, and design automation toolchains open the path to apply iterative development cycles and reproducibility of results. Moreover, usability layers that span from the hardware-software interfacing to ways of programming the architecture are necessary for a user base creation and deploying usable hardware. For these reasons, this dissertation explores these crucial issues and presents relevant takeaways of the domain specialization role of reconfigurable computing systems. It begins with two Chapters that provide a bird's-eye view of the latest reconfigurable computing trends and the main ways of programming FPGAs and how to interact with them. Then, the dissertation dives into two relevant computing domains: image registration and regular expressions. For each domain, the dissertation includes design methodologies, design automation, and usability layers (with a particular focus for each of them), three critical aspects of implementing a DSRA.

Sommario

Il campo delle architetture dei calcolatori sta affrontando ostacoli tecnologici e architetturali che limitano la scalabilità dei processori generici nelle prestazioni fornite ad un costo energetico ragionevole. Pertanto, i progettatori di architetture devono seguire nuovi percorsi per ottenere calcoli più efficienti dal punto di vista energetico con la tecnologia attualmente disponibile, ad esempio impiegando soluzioni *specializzate in particolari domini*. In effetti, le architetture specializzate nel dominio possono fornire prestazioni estremamente elevate con un profilo energetico relativamente basso e anche di più se combinate con astrazioni di alto livello per la progettazione e la programmazione. Le architetture specifiche del dominio (DSA) sono generalmente, oltre all'esponente di spicco per la specializzazione del dominio, architetture programmabili a livello software progettate per essere implementate in modo efficiente come circuiti integrati specifici per le applicazioni (ASIC) o parte di Sistema su un Chip (SoC), per pochi compiti, e con la minima quantità di tecniche microarchitetturali avanzate basate su CPU. Tuttavia, lo sviluppo di dispositivi in silicio personalizzati è un processo lungo e molto costoso che non è sempre compatibile con il tempo per andare sul mercato e la rapida evoluzione delle applicazioni. Pertanto, le piattaforme informatiche adattabili rappresentano l'alternativa più praticabile per questi scenari. Gli FPGA (Field-Programmable Gate Arrays) sono le piattaforme candidate per il loro tessuto eterogeneo riconfigurabile sul *campo*. Oltre alla riconfigurabilità, gli FPGA possono implementare grandi architetture di calcolo spaziale e sono disponibili pubblicamente su piattaforme di calcolo cloud.

Pertanto, questa tesi si concentra sulle *Architetture Riconfigurabili a Dominio Specifico* (DSRA): architetture specializzate nel dominio con datapath *adattabili* implementate su FPGA. La progettazione di tali architetture richiede una visione chiara delle tendenze a livello di sistema e dei livelli di astrazione degli FPGA. Inoltre, le metodologie di progettazione avanzate consentono architetture efficienti dal punto di vista energetico su misura per il dominio e le toolchain di automazione della progettazione aprono la strada all'applicazione di cicli di sviluppo iterativi e alla riproducibilità dei risultati. Inoltre, i livelli di usabilità che vanno dall'interfaccia hardware-software alle modalità di programmazione dell'architettura sono necessari per la creazione di una base di utenti e l'implementazione di hardware utilizzabile. Per queste ragioni, questa tesi esplora questi argomenti cruciali e presenta importanti spunti sul ruolo dei sistemi di calcolo riconfigurabile nella specializzazione del dominio. Inizia con due capitoli che forniscono una panoramica delle ultime tendenze del calcolo riconfigurabile e delle principali modalità di programmazione degli FPGA e di come interagire con essi. Quindi, la tesi si tuffa in due domini di calcolo rilevanti: registrazione di immagini ed espressioni regolari. Per ogni dominio, la tesi includerà metodologie di progettazione, automazione della progettazione e livelli di usabilità (con un focus particolare per ciascuno di essi), tre aspetti chiave dell'implementazione di un DSRA.

Contents

CHAPTER 1

Introduction

In recent years, there has been a shift in the computer architecture landscape. Moore's Law [2] and Dennard's Scaling [3] have driven for almost 50 years the research on general-purpose architectures (i.e., Central Processing Units (CPUs)) and their optimizations, thanks to scaling in transistor count and power budget, respectively. However, the end of these laws [4–8] pushes the researchers close to the boundary for technological and architectural reasons. Figure 1.1 shows 48 years of trends in commercial microprocessors for relevant metrics: from the number of cores within each chip to the achieved operational frequency, from the single-thread performance to transistor count, and the typical power budget. The reader can notice how the trends reach a plateau for each considered metric, showing a scaling interruption. Similarly, Figure 1.2 focuses on the performance improvement based on the SPECint CPU benchmark and better highlights the trends in the general-purpose processor field. In general, the charts clearly show that the performance improvements offered by the CPUs can not rely anymore only on technological advancements such as transistor doubling by Moore's Law or the constant power consumption for the same chip area by Dennard's scaling. Hence, computer architects have to consider a new way to conceive architectures, for example, by narrowing the

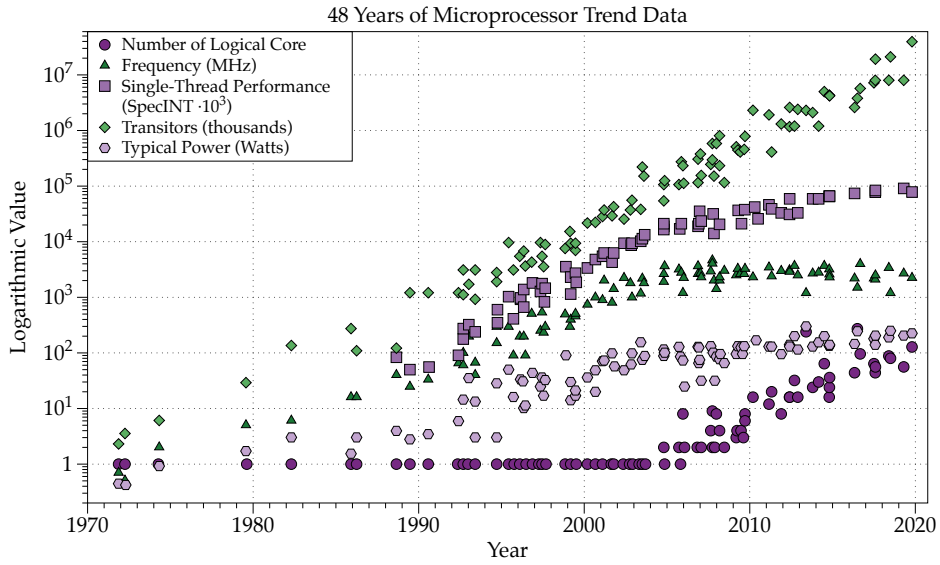


Figure 1.1: *48 Years of Microprocessor Trend Data [1] (logarithmic scale). Original Data (up to 2010) from M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten. Data after 2010 from K. Rupp.*

domain of the targeted computations. This domain narrowing is one of the most promising approaches to harvest more energy-efficient calculations unless a new disrupting technology appears on the panorama [5]. For these reasons, the current architectural focus is shifting from general-purpose to domain-specific engines.

In this scenario, the *domain specialization* path builds on a comprehensive environment where hardware and software are both specialized towards a particular application domain rather than being general purpose [8]. On the one hand, the software-centric approaches devise at their core the so-called Domain-Specific Languages (DSLs). DSLs offer a straightforward approach to the considered domain while retaining code portability and delivering noteworthy performance. Usually, DSLs leave the burden of optimizing the overall process, including tailoring the code to the underlining architecture, to extremely powerful compilers, which may not always be enough to deliver the needed performance. On the other hand, hardware-centric approaches focus on conceiving domain-specialized architectures to improve execution time and energy efficiency from Application-Specific Integrated Circuit (ASIC) to Application-Specific Instruction-set Processor (ASIP), from Domain-Specific Architecture (DSA) to Coarse Grain Reconfigurable Architecture (CGRA) or even simple domain accelerators.

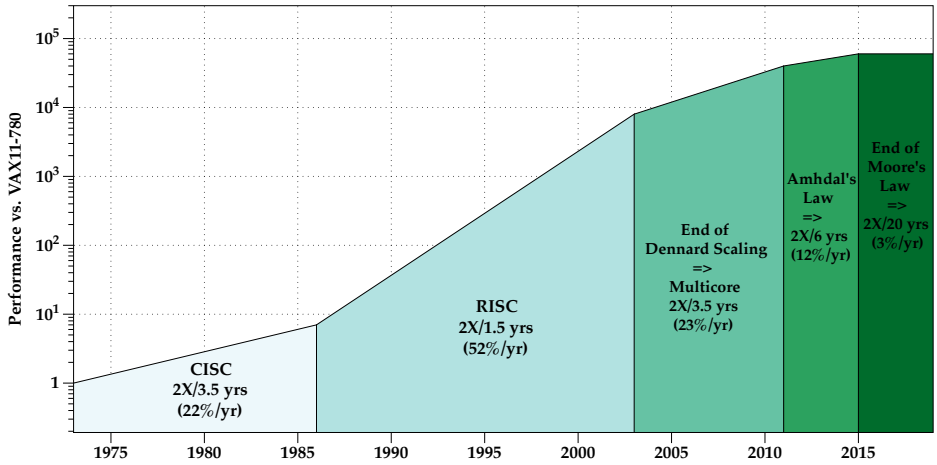


Figure 1.2: *About 50 Years of processor performance improvement based on the SPECint CPU benchmark. Image adapted from [9] based on [5].*

Among these approaches, DSAs perform only a few tasks extremely efficiently, rather than being designed for general workloads, and are often combined with an open software ecosystem [5, 6]. DSAs are considered as the primary path towards the domain specialization [5,8,10–13], even more, if combined with the usage of a DSL. The combination of hardware- and software-centric approaches delivers skyrocketing gains against general-purpose processors, which are slowly facing an insurmountable obstacle (i.e., technological and architectural reasons). A DSA usually is software-programmable, hence it presents an abstraction layer such as an Instruction Set Architecture (ISA), Turing complete, and employs the easiest yet advanced computer architecture techniques to build a fixed datapath. In this way, a DSA generally comprehends the most straightforward data type and size, as well as the most suitable form of parallelism required by the domain. Moreover, it employs the simplest microarchitectural techniques and the most fitting memory hierarchy such that designers can exploit saved area for bigger functional units or memories. These are the general DSA guidelines presented by Hennessey and Patterson during their Turing Award Lecture [8] towards the design of ASICs-based DSAs [5]. Nonetheless, for a long time, developing a full custom ASIC has been a time-consuming process where the time to market and the Non-Recurring Engineering (NRE) costs were, and currently are, critical and not negligible. Indeed, in this scenario, productivity becomes a crucial point to keep pace with the fast-evolving nature of the applications. For these reasons, both companies and

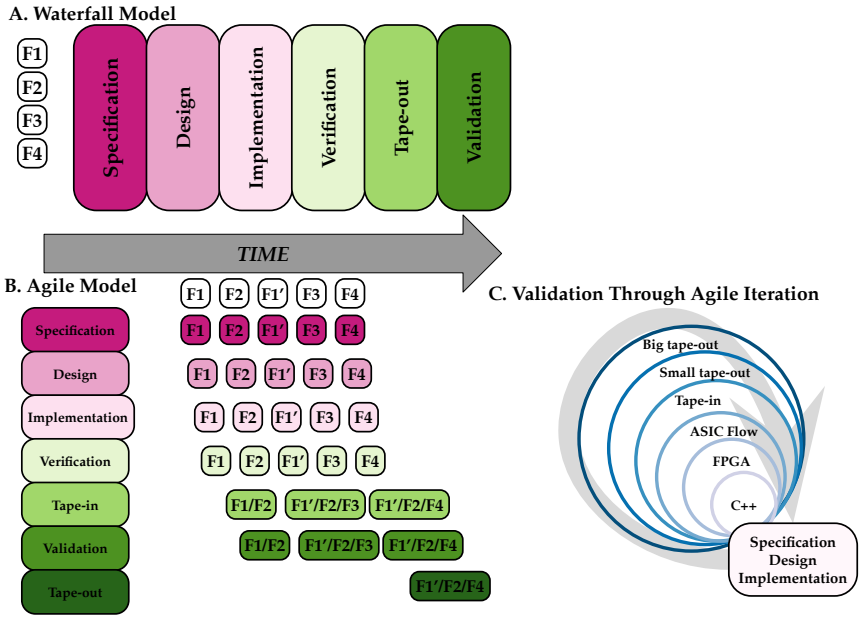


Figure 1.3: *Contrasting the agile and waterfall models of hardware design for desired feature(s) F#. Adapted from [14].*

the academic world are trying to apply iterative approaches (e.g., Agile represented in Figure 1.3-B./C.) to fast hardware development [14–16] along with a software stack for complete usability for both design and communication perspectives [17, 18]. In this scenario, adaptable computing platforms are becoming essential for iterative hardware lifecycles, dynamic workloads, and the composable data center infrastructure [19].

Field Programmable Gate Arrays (FPGAs) are the state-of-the-art platform for Reconfigurable Computing (RC) and the foundations of the adaptable computing paradigm [20]. They can provide a generic system with a heterogeneous fabric, reprogrammable at the circuit level after manufacturing along with comprehensive software development tools and Application Programming Interfaces (APIs). Traditionally, FPGAs were employed only for glue logic design, ASIC prototyping, sensor fusion, and fields, such as the telecommunication one, where field-(re)programmable architecture (i.e., adaptable after deployment) are crucial [21]. However, thanks to their spatial computing characteristics, FPGAs, and more in general RC systems, have proven their effectiveness against CPUs in terms of performance and energy efficiency [22–31], and their acceleration potential even in a large scale data-center infrastructure [32, 33]. All these reasons, along with the

public availability of FPGAs in the cloud [32, 34–37], make these devices the candidate platforms for devising fast-evolving domain-specialized architectures without incurring in ASIC NRE costs and providing adaptable computing platforms thanks to the heterogeneous reconfigurable fabric.

1.1 Problems Statement

Despite being the commercial platform closest to the ideal adaptable computing paradigm, FPGAs (and all reconfigurable systems) deserve a deeper analysis of their role in the domain specialization path. Indeed, they can implement domain-specialized architectures that can be updated after field deployment, delivering variable datapaths which are adaptable almost an infinite number of times. These architectures are called in this dissertation *Domain-Specific Reconfigurable Architectures (DSRAs)*.

Employing RC systems, such as FPGAs, opens a wide variety of *architectural organizations* (different from traditional CPUs with their fixed datapath) to design DSRAs, and requires a precise classification. However, one of FPGAs' drawback has always been their programmability process, that usually requires low-level abstractions and system-level knowledge, which is incredibly time-consuming. Hence, having a clear view of how to *design* these domain-specialized architectures is paramount to deliver a design process that keeps pace with applications' quickly evolving nature and exploit ready-to-use *programmability abstractions*.

In these regards, the language and the design methodology for DSRAs represent key elements. The aforementioned design guidelines [8] are crucial also for the design of DSRAs, but their employment introduces design issues. Some of them are shared with the ASIC ecosystem whenever considering a fixed datapath, and others are inapplicable given the deeper ASIC physical-design requirements and post-silicon verification, rarely applied in the FPGA-based designs. For instance, frequently *adapting a DSRA datapath* to the most simple and reduced precision, or employing different arithmetic, for a given computational domain, might introduce unconsidered problems, incurring efficiency worsening and not domain-specialization improvements. Additionally, particular domains may present more than a single *form of parallelism* or *computational pattern* that fits the design process of a domain-specialized architecture and different applications. As an example, the Regular Expressions (REs) domain, which is intrinsically sequential, can adopt both a *depth-first* or a *breadth-first* execution model for a DSRA. Therefore, exploring the computational models for the same domain is paramount to devise an effective architectural solution and regu-

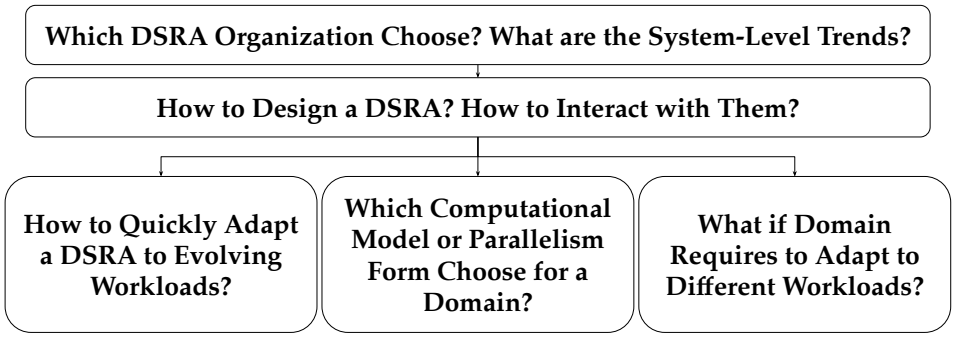


Figure 1.4: *Graphical representation of this dissertation considered problems.*

larly *adapt* to the considered *workloads*. Finally, the architecture software-programmability is another remarkable issue to build a success DSRA (as well as for ASIC-based DSA), grant a user-base, and build a community.

On top of these issues, DSRAs, or even traditional FPGA-based accelerators, require the support of Computer Aided Design (CAD) toolchains or high-level generators to enable *the usability* of the architectures, the deployment to different FPGAs, and an *automatic process to adapt* to the changing workload. Without such automation toolchains, designers can not exploit the intrinsic adaptability of reconfigurable systems for fast and iterative hardware developments and deployments as well as fast prototyping purposes. Figure 1.4 summarizes the problems considered in this dissertation.

1.2 Contributions

This dissertation defines and analyzes specialized computer architecture organizations based on reconfigurable platforms called DSRAs. Starting from an analysis of the latest system-level trends and ways of programming FPGAs, I will address three main topics in specific domains: design methodologies, automation, and usability. The first one (i.e., the design methodologies) is crucial for designing highly energy-efficient architecture; while automation is essential for fast iterative approaches to newer solutions development and reproducibility of achieved results; the last one (i.e., usability) comprehends software programmability in a complete view that spans from hardware-software interfacing to ways of programming the architecture. This dissertation builds on top of these three main topics and presents them comprehensively for each discussed work.

Generally, the first yet most crucial task for domain-specialized architec-

tures design methodologies is domain identification. This dissertation will consider as domain a class of algorithmic problems that share computational or memory access patterns. Then, starting from John Hennessey and David Patterson's general design guidelines on DSAs [5], I identify three prominent architectural organizations of DSRAs and classify them depending on two orthogonal characteristics: level of software programmability and datapath configurability. The first architecture organization, the most traditional reconfigurable DSA[1], is based on a "fixed" datapath with a dedicated ISA that communicates with instructions and data memories. The second one, the so-called streaming architecture, has a fixed datapath created for each class of problems, generally devised from a high-level tool that automates the whole process. Finally, the third architecture organization is a hybrid version between traditional DSAs and streaming architectures with a semi-fixed datapath, i.e., a CGRAs [38]. Being CGRAs mainly a theoretical platform that is either simulated or leverages FPGAs [38], although there are many interesting research efforts [38, 39], I will focus on traditional DSAs, similar to general-purpose processors and streaming architectures. If the reader is interested in domains widely explored in these directions, Machine Learning and Neural Networks domains present several examples of these architectures [6, 8, 11, 40–42], though a scalability study on multi-FPGA devices has been done only recently [43].

Within the context of specialization, the domain and its workloads are among the top driving factors for designing a DSRA. Inspired by Asanovic et al.'s work, where they identified thirteen *dwarfs*, or key computational kernels, and discuss the full-spectrum of the computing landscape [44], I found relevant computational kernels that deserve an in-depth analysis: linear algebra and finite state machines. This dissertation analyzes two main domains with computations affine to some of the dwarfs identified as key computational kernels [44]: *image registration* (an imaging technique based on linear algebra computations for optimization methods), and *regular expressions* (finite state machines).

Image Registration (IRG) is an essential pre-processing step of several image processing pipelines. However, it is often neglected for its context-specific nature [45–47]. I present and discuss an open-source framework that provides a domain-specific streaming-based architecture [30]. The proposed design methodology based on a dataflow MapReduce computations enables the engineering of a highly customizable Mutual Information (MI) accelerator [30], one of the most computationally intensive parts of IRG procedures. In this way, the IRG procedure is extremely optimized from a

[1] From now on, the term DSA is used to refer to the first category of the identified DSRAs

performance and energy efficiency perspective.

The regular expression domain falls in the Finite State Machine (FSM) dwarf kernel. While the literature provides several exciting approaches for streaming/in-memory architectures, what we are missing today is a DSA able to tackle this problem with high performance and flexibility in the pattern to search. For these reasons, I provide an extensive analysis of DSA for the regular expression domain, showing two different architectures that provide different levels of solutions. These two architectures explore different computational models based on a depth-first like approach [48] and one theoretically expressed by Russel Cox based on a breadth-first approach [49].

Additionally, designing DSRAs, or architectures in general, requires a CAD infrastructure to support replicability, fast exploration, and quick iterations. For these reasons, all the proposed architectures exploit a shared design automation methodology for easing their replicability and verification.

In summary, the contributions of this dissertation are:

- an analysis of the latest reconfigurable system-level trends with a taxonomy of domain-specific reconfigurable computer organizations;
- a survey with taxonomies and timelines of the most prominent digital design abstractions for FPGAs;
- an open-source design automation framework for highly customizable streaming-dataflow domain specialized accelerators proven on the IRG domain;
- an exploration of different computational model and form of parallelism for the REs (or equivalently Finite State Machines) domain for traditional DSAs;

1.3 Sources

This dissertation refers to and possibly extends the following publications:

- "Reconfigurable architectures: the shift from general systems to domain specific solutions". D'Arnese, E., **Conficconi, D.**, Santambrogio, M. D. and Sciuto, D., Springer Book chapter [18];
- "Pushing the Level of Abstraction of Digital System Design: a Survey on How to Program FPGAs". Del Sozzo, E.,**Conficconi, D.**, Zeni, A., Salaris, M., Sciuto, D., and Santambrogio, M. D. (Submitted to ACM Computing Surveys (CSUR) [17]);

- "Optimizing bit-serial matrix multiplication for reconfigurable computing". Umuroglu, Y., **Conficconi, D.**, Rasnayake, L., Preusser, T. B., and Själander, M. 2019. ACM Transactions on Reconfigurable Technology and Systems (TRETS) [50];
- "A Framework for Customizable FPGA-based Image Registration Accelerators". **Conficconi, D.**, D'Arnese, E., Del Sozzo, E., Sciuto, D., and Santambrogio, M. D. 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA21), Awarded with ACM Artifacts of Available, Reusable, Reproduced [30];
- "TiReX: Tiled regular expression matching architecture." Comodi, A., **Conficconi, D.**, Scolari, A., and Santambrogio, M. D. In 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) [48];
- "An Energy-Efficient Domain-Specific Architecture for Regular Expressions." **Conficconi, D.**, Del Sozzo, E., Carloni, F., Comodi, A., Scolari, A., and Santambrogio, M. D. (Submitted to IEEE Transactions on Emerging Topics in Computing (TETC)) [51];
- "CICERO: A Domain-Specific Architecture for Efficient Regular Expression Matching." Parravicini, D., **Conficconi, D.**, Del Sozzo, E., Pilato, C., and Santambrogio, M. D. 2021. ACM Transaction on Embedded Computing Systems (TECS) as part of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) proceedings [52];
- "Enhancing the Scalability of Multi-FPGA Stencil Computations via Highly Optimized HDL Components." Reggiani, E., Del Sozzo, E., **Conficconi, D.**, Natale, G., Moroni, C., and Santambrogio, M. D. (2021). ACM TRETS [43];
- "Dovado: An Open-Source Design Space Exploration Framework." Paletti, D., **Conficconi, D.**, and Santambrogio, M. D. 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) [53];

1.4 Thesis Organization

The dissertation Chapters' division is organized to reflect Figure 1.4 problems and ease the reader access to the content based on a specific domain or a given architectural organization.

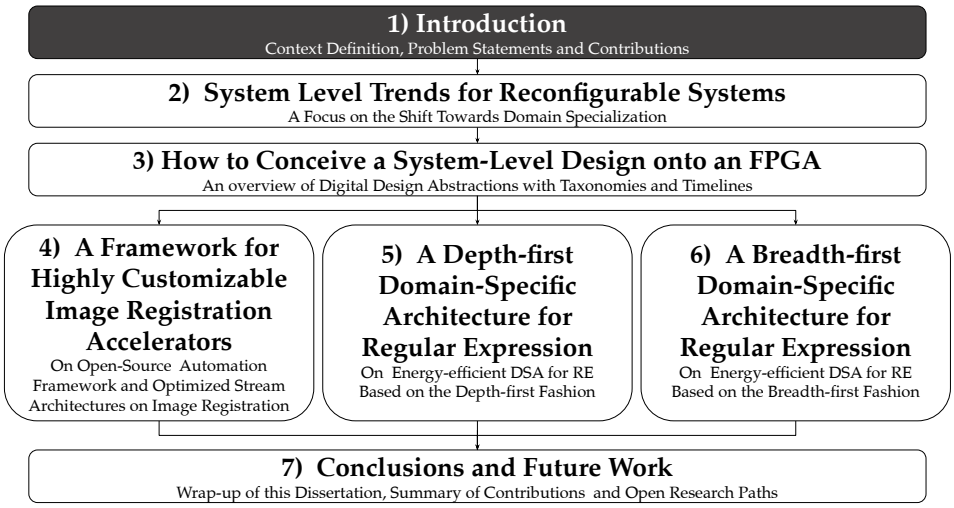


Figure 1.5: *Graphical dissertation outline: Chapter 2 and 3 describe the main trends towards domain specialization of Reconfigurable Systems; then the dissertation describes two computational kernels dissected in different aspects of the specialization (design automation, computational pattern); finally, it concludes, focusing on open research paths.*

As Figure 1.5 highlights, the organization is the following.

- Chapter 2 provides background on FPGA architectures and analyzes the latest system-level trends of reconfigurable computing systems.
- Then, Chapter 3 presents the three relevant digital abstractions for FPGAs surveying the main exponent of each abstraction according to a proposed taxonomy and a timeline.
- Chapter 4 presents an open-source framework for highly customizable streaming-dataflow domain specialized architecture for MI calculus and IRG.
- Chapter 5 and Chapter 6 analyze two traditional DSAs employing different execution models based on a depth- or breadth-first approach respectively.
- Finally, Chapter 7 concludes this dissertation while highlighting the most important takeaways.

CHAPTER 2

Reconfigurable architectures: the shift from general systems to domain specific solutions

This Chapter analyzes the system-level trends of reconfigurable architectures starting from the early democratization efforts with the abstraction level rise to the recent domain specialization. It begins with brief background knowledge on Field Programmable Gate Arrays (FPGAs)' components. Then, it depicts the never stopping efforts on FPGAs' democratization through the constant development of design automation toolchains and newer abstraction layers, which overall ease the design development and their usage. Afterward, it shows the main recent trends in which Reconfigurable Computing (RC) saw its explosion. Examples are the public cloud availability, which enables the paradigm of Hardware-as-a-Service for these platforms, and the increasing heterogeneity even at the system level. As one of the most exciting newer trends for RC-systems, there is the domain specialization one, in which this Chapter depicts a possible taxonomy of Domain-Specific Reconfigurable Architectures (DSRAs)' organizations. Finally, the Chapter concludes with open questions and future research directions of system-level trends.

Chapter 2. Reconfigurable architectures: the shift from general systems to domain specific solutions

Reconfigurable computing is an expanding field that, during the last decades, has evolved from a relatively closed community, where hard-skilled developers deployed high-performance systems, based on their knowledge of the underlying physical system, to an attractive solution to both industry and academia. With this Chapter, we explore the different lines of development in the field, with a specific focus on those based on Field Programmable Gate Arrays (FPGAs), namely the need for new tools to shorten the development time [25], the creation of heterogeneous platforms that couple hardware accelerators with general-purpose processors [54], the differentiation of the paradigms employed and applicative scenarios [55], and the demand to move from general to domain-specific solutions. Starting with the identification of the main limitations that have led to improvements in the field, we explore the emergence of a wide range of Computer Aided Design (CAD) tools that allow the use of high-level languages and guide the user in the whole process of system deployment. This opening to a broader public and their high performance with relatively low power consumption facilitate the spreading in data-centers, where, apart from the undeniable benefits, we have explored critical issues. We conclude with the latest trends in the field, such as using hardware as a service and shifting to Domain-Specific Architectures (DSAs) based on reconfigurable fabrics, i.e., Domain-Specific Reconfigurable Architectures (DSRAs).

Considering the magnitude of the topics, we will cover the time-span between a period when only a restricted elite of people knew and exploited reconfigurable systems and the current days where they are often integrated into data centers and provided as services to a broader audience. Following this path, we can identify three main trends that helped the adoption of reconfigurable fabrics, specifically FPGAs, in a variety of computation systems: a programming paradigm democratization, a development of heterogeneous platforms, and new accessibility solutions. In a first instance, reconfigurable fabrics were primarily employed in telecommunications thanks to the possibility of easy and fast reconfiguration and signal processing for speeding up the computation of specific algorithms [56]. At that time, the deployment time was quite long and limited to a few skilled developers [57]. Although the computational benefits were undeniable compared to general processors [22, 23], the development bottlenecks limited such platforms' potential. For these reasons, a considerable effort was put in developing toolchains and abstraction layers to help developers, not necessarily hard-skilled on the topic, to benefit from reconfigurable fabrics [58–61]. The second attempt to increase the usage of reconfigurable systems by a larger number of users was combining them with general-

purpose processors and, later on, with software-programmable vector engines. The coupling with micro-controller and hard-processors opens to different applicative scenarios but also introduces new challenges on interconnections and memory coherency [54]. Indeed, the aforementioned heterogeneity and high connectivity favor the adoption of reconfigurable systems in the cloud computing ecosystem, where the power wall hit with the dark silicon [4] makes the providers craving for energy-efficient solutions, such as reconfigurable systems. Finally, the newest trend in the reconfigurable system field is to further promote their use by users closer to the software world. In this sense, we are witnessing an increasing number of providers of reconfigurable platforms in the cloud as services for the final users [55]. Combined with the opening to the broader public, various efforts have been put in the development of DSAs, which enable the user to develop applications that run on a reconfigurable system using a Domain-Specific Language (DSL).

Based on all these considerations, our review will start by describing background knowledge on FPGAs architectures (Section 2.1) and the State of the Art around 2010, highlighting the main limitations that pushed for the improvements that we have anticipated (Section 2.2). The reader interested in dynamic reconfiguration and low-level technological details can refer to [26, 62–64], or for detailed trade-offs analysis in Reconfigurable Computing (RC) systems [22–26]. Hence, following the different lines of development in the past ten years (Section 2.3), we will end by describing the current trends with their paradigm shift (Section 2.4) and, finally, the possible trends we will see in the following years (Section 2.5). Figure 2.1 summarizes the main system-level trends this Chapter deals with: from more structured CAD tools to the rise of heterogeneous architectures and the shift to new paradigms such the FPGA-as-a-Service (FaaS) and the spread of DSRA in the domain specialization direction.

2.1 Background on FPGA Architecture

FPGAs are an array of heterogeneous programmable blocks (e.g., logic, storage, high-speed arithmetic, hardened memory controllers, I/O) that can be flexibly connected through routing switches usually based on Static Random Access Memory (SRAM)[1]. Figure 2.2 shows an example of FPGA architecture of the early days, in which Xilinx pioneered FPGA in 1984, and modern heterogeneous architecture with multiple hardened blocks. FPGAs

[1] Several other FPGA technologies exist, and the interested reader can have a look an overview of them at other references [25, 26, 65].

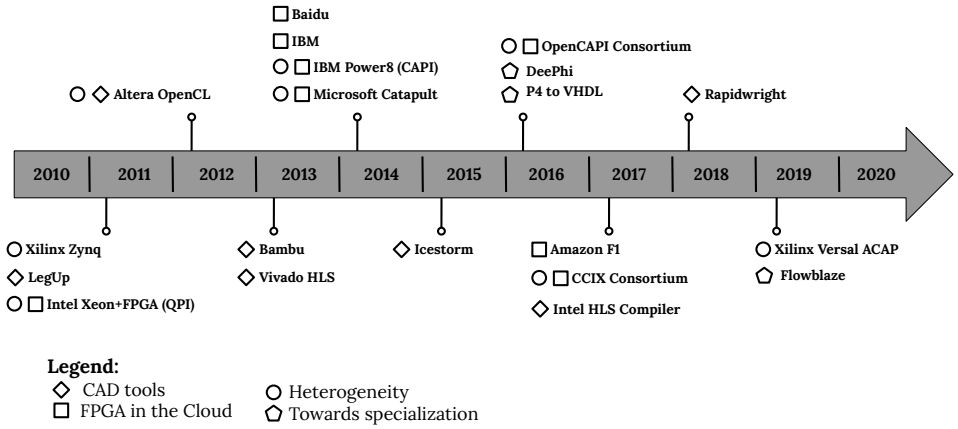


Figure 2.1: *Timeline of a selection of some relevant works following the taxonomy proposed in this Chapter.*

are called *field-programmable* since they can be configured after manufacturing an almost infinite amount of times to implement any digital hardware design. The users describe the desired functionality in several digital abstractions, such as Hardware Description Languages (HDLs), High-Level Synthesis (HLS), and DSLs (more details in Chapter 3), which CAD toolchains then translate into a bitstream to configure the FPGA. These toolchains synthesize HDL designs into a circuit netlist, map them to heterogeneous fabric blocks, and route their connections through the routing layer. Employing a ready-to-use FPGA to implement a system enables the lowering of Non-Recurring Engineering (NRE) costs and the shortening of time-to-market (by skipping physical design, layout, fabrication, and verification chips stages) against an Application-Specific Integrated Circuit (ASIC). In this way, FPGAs represent the candidate solution for medium (or small) volumes and fast-paced product cycles markets. Moreover, their adaptable reconfigurable fabric avoids the drawbacks of fixed systems such as Central Processing Units (CPUs) and Graphics Processing Units (GPUs). It enables designers to achieve better efficiencies from both the execution times and energy efficiency perspectives. However, FPGAs' flexibility delivers worst area and frequency achievements compared to a full custom ASIC while retaining incredibly higher adaptability.

2.1.1 Programmable Logic

The base FPGA architecture builds on top of an array of Look-Up Tables (LUTs) superseding traditional Complex Programmable Logic De-

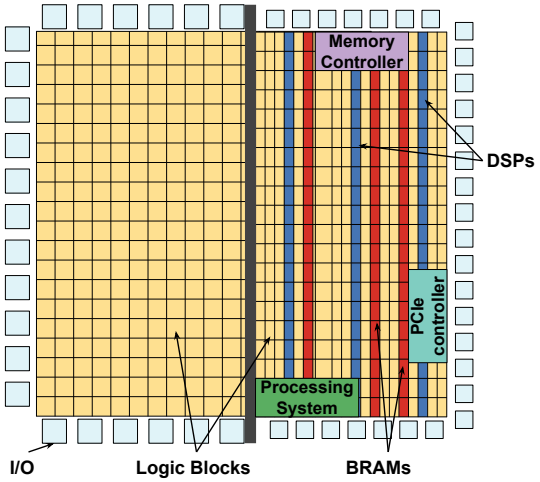


Figure 2.2: *Early FPGA architecture with programmable logic and I/Os against modern heterogeneous ones with BRAMs, Digital Signal Processings (DSPs), and other hardened components.*

vices (CPLDs), Programmable Logic Arrays (PLAs), Programmable Array Logics (PALs) technologies. Moreover, it contains Flip Flops (FFs) to provide small local storage and adders for simple mathematical functionalities. A ***K*-LUT** can implement boolean functions with K-input. It stores the truth table of desired functionality and selects one of the 2^K output values by employing the K input as multiplexer select signals. A Basic Logic Element (BLE)[2] traditionally constitutes of a K-LUT, an output register, and a bypassing multiplexer [64]. In this way, BLE implements either a K-LUT with registered or unregistered output (thanks to the multiplexer) or a FF. Multiple BLEs cluster in Logic Blocks (LBs) with its local interconnect within BLE outputs or LB inputs and a target destination. Therefore, an LB contains N BLEs of K-LUTs. Increasing K enables more functionalities packed in a single LUT, reducing the critical path while increasing the area exponentially and decreasing the speeds linearly. Moreover, increasing N decreases the demand for inter-LB routing at the cost of increasing the area quadratically and degrading linearly the speeds with the number of BLEs (i.e., N) [64][3]. A significant improvement comes with the introduction of fracturable LUTs by Altera in 2003 with the Stratix II [66] that combines the performance of larger LUTs with the area-efficiency of smaller LUTs. A fracturable $\{K, M\}$-LUT can implement a single K-LUT or two $K-1$

[2] The term BLE is general enough to cover at least both Intel and Xilinx terminologies of the smallest configurable element Adaptive Logic Module (ALM) and Configurable Logic Block (CLB) respectively. [3] The first Xilinx LUT-based FPGA in 1984 had an LB with two 3-LUTs (i.e., $N = 2$, $K = 3$).

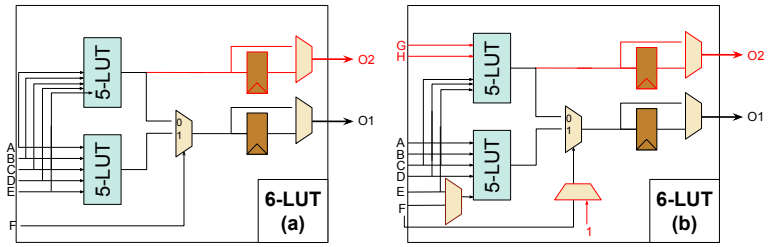


Figure 2.3: *6-LUT fracturable into two 5-LUTs with (a) no additional input ports (5 shared inputs) or (b) two additional input ports and steering multiplexers (3 shared inputs). Image adapted from [64].*

LUTs that use at most $K + M$ distinct inputs. Figure 2.3 shows the structure of a fracturable 6-LUT built with two 5-LUTs and a 2:1 multiplixer. Early non-fracturable LUT were coupled with a single **FF**. However, the fracturable LUTs add a second FF to register both outputs. Arithmetic operations, such as addition and subtraction, are widespread in FPGA-based designs. Although they can be implemented with LUTs, each ripple carries adder's bit requires two LUTs (one for the carry and one for the output), increasing the utilization and slowing the critical path. Therefore, modern FPGAs include **hardened arithmetic** in their LBs. They reuse LUT routing ports or LUT outputs to avoid expensive routing ports. Moreover, the carry bits have specific interconnect with almost no programmability for a fast carry path. Indeed, Xilinx Versal hardens the carry logic for 8-bit carry look-ahead adders (enabling an addition every eighth BLE) [67], while Intel Agilex delivers 2 arithmetic bit per BLE and a dedicated carry interconnect [68].

2.1.2 Programmable Routing

The programmable routing is an essential part of the reconfigurable fabric of pre-fabricated wiring segments and programmable switches to connect any function block. Hierarchical and island-style FPGAs are the two main routing classes. However, hierarchical classes suffer from very long wires connections that become troublesome with the node process scaling. The island-style (shown in Figure 2.4) is inspired by the regular structure of the 2D layout with horizontal and vertical wires. It includes routing wire segments, connection blocks for logic-routing wires connection, and switch blocks to form longer routes by connecting routing wires. A good routing architecture has to balance the number of programmable switches and wiring segments and their area costs. Indeed, modern routing fabrics con-

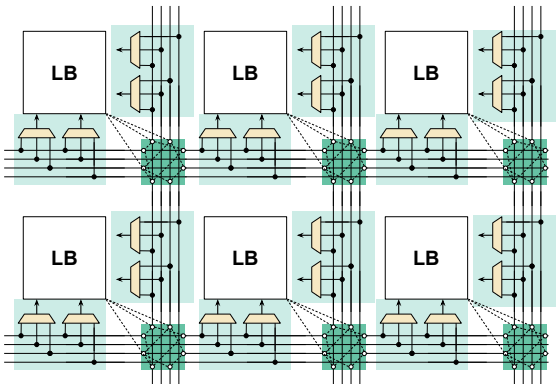


Figure 2.4: *Island-style routing architecture. Thick solid lines are routing wires while dashed lines are programmable switches. Connection and switch blocks are shaded in light and dark green, respectively. Image adapted from [64].*

tain wiring segments of multiple lengths (both short and long) to match different application requirements. For instance, Intel's Stratix FPGAs create a sort of hierarchical structure in an island-style FPGA since they enable long wires connection only with short ones [69].

2.1.3 Programmable I/O

Traditionally, FPGAs were used as glue logic components or sensor fusion platforms for the presence of programmable I/O banks. Indeed, these I/O interfaces can flexibly adapt to different standards, electrical characteristics, voltage levels, timing requirements, and protocols, and they occupy a significant amount of FPGA's area. These I/O blocks generally contain additional hardened circuitry to simplify communication between the outer world and the reconfigurable fabric (e.g., capture registers, double- to single-data rate conversion registers, serial-to-parallel converters). Modern FPGAs now contain high-speed I/O banks to implement serial protocols, such as Ethernet and PCIe, that embed the clock in data transitions and run at about 28 Gb/s. Overall, also the FPGA I/O architecture must balance the high speeds required with the flexible programmability purpose.

The data exchange between different chips or boards is divided into two main transmissions types: parallel and serial. In the former, data is sent simultaneously from a Transmitter (TX) to a Receiver (RX) on separate communication lines and synchronized with a common clock. In the latter, data are bit-wise serialized and transmitted sequentially on the same channel. The communication can be either synchronous when the transmission clock is provided by RX and sent to TX, or asynchronous, when TX trans-

mits data asynchronously to RX (i.e., there is not a communication media containing a synchronization clock), which takes care of recovering data and clock. Historically, parallel buses were the standard for chips communication since they avoid the overhead of serialization/deserialization. However, the ever-increasing bandwidth demand imposes bigger parallel data buses and clock rates, which, along with signal degradation over long distances, made room for serial transmission. Therefore, serial transmissions have become advantageous for off-chips, board communications, as they can provide higher performance while minimizing pin counts, simultaneous output switching noise, and overall system costs. A perfect data transmission among TX and RX interfaces would require an ideal clock synchronization in terms of both frequency and phase. Hence, a Synchronous Serial Interface may seem the way, as the same clock is forwarded from RX to TX. However, non-idealities introduced by the communication media add jitter, distortions, and attenuation to the transmitted clock.

FPGAs include dedicated hardware for serial links, known as Serializer/Deserializer (SerDes), which provides a high-speed Asynchronous Serial Interface. In this context, Multi-Gigabit Transceivers (MGTs) are special SerDes capable of reaching bit rates above 1 Gigabit/second, used for for high-speed data communications. The MGT main building block is a Parallel In Serial Out (PISO)/Serial In Parallel Out (SIPO) couple, where parallel data coming from the data bus are serialized by the PISO and transmitted through a communication media to the SIPO, which is in charge of recovering serial data in its original form. A method to correctly recover data at the receiver side of the transmission link is the Clock Data Recovery (CDR) circuit. This circuit creates a recovered clock that is phase-aligned with the input transition position and samples the incoming data stream. The CDR is usually composed of a Phase Locked Loop (PLL), a negative feedback circuit that, through a Voltage Controlled Oscillator (VCO), generates an output frequency locked to the incoming data streams in both frequency and phase. The Phase Detector (PD) output a signal proportional to the phase difference of its inputs, averaged by a low-pass filter, and, finally, the VCO generates a clock whose frequency is proportional to its input amplitude. The negative feedback loop reacts to generate a stable recovered clock, which is locked to the input stream in phase and frequency. To ensure correct CDR operations, enough transitions are paramount, thus requiring the usage of line encoding scheme such as 8b/10b or 64b/66b. Thanks to their high working frequency, modern MGTs allow extremely high-throughput communications and are widely used to interconnect FPGAs [32, 33, 36, 70, 71].

Table 2.1: *Different reading/writing ports BRAMs configuration and their required routing ports number (W: data width, D:BRAM depth).*

BRAM Ports	Mode	#Routing Ports
1read	Single-port ROM	$log_2(D) + W$
1read/write	Single-port RAM	$log_2(D) + 2W$
1read+1write	Simple dual-port RAM	$2log_2(D) + 2W$
2read/write	True dual-port RAM	$2log_2(D) + 4W$
2read+2write	Quad-port RAM	$4log_2(D) + 4W$

2.1.4 On-Chip Memory

As highlighted in Section 2.1.1, the first on-chip memories were FFs within LB. Nevertheless, with the growing implementable systems on FPGAs and the continuously growing logic capacity, more data locality became needed, and LUT-based Random Access Memory (RAM) were inefficient. Moreover, there is not a universal RAM configuration (in terms of capacity, word width, number of ports) in FPGA-based designs. Therefore, Altera introduced in 1995 the Block RAMs (BRAMs) [72]. Typically, a BRAM builds on an SRAM-based memory with additional modules to configure the component and connect to the routing fabric. Balancing the BRAMs' capacity, data width, and the number of read/write ports means to control the linear area usage of SRAM cells and the sublinear growth of the rest of the components. Moreover, ever-changing application needs demand for new components to configure BRAMs' width and depth through multiplexing circuitry [73–76]. BRAMs present an interface toward the programmable routing similar to the one of the LBs to favor regular interconnections. Depending on requirements of reading/writing ports, the BRAMs can be configured accordingly, as shown in Table 2.1. Interested designers can exploit as another source of on-chip memories the *K*-LUTs, which are native 2^K 1-bit read-only memories, by adding the missing write circuitry. These memories are called LUT-RAMs. Usually, the vendors make eligible only 50% of the logic fabric to LUT-RAMs, since, for efficiency purposes, it requires an entire LB for a LUT-RAM.

Nowadays, vendors employ a medium size of BRAMs, e.g., Intel FPGAs employ 20kb BRAMs, while Xilinx ones employ 18kb BRAMs (combinable two by two to form 36kb memories) However, there is no general agreement on the BRAMs' size since, for example, Xilinx high-end devices include large 288kb memories called Ultra RAMs (URAMs).

2.1.5 DSP Blocks

The initial arithmetic circuits in commercial FPGAs were the additional logic of adders and carry chains. Therefore, multiplication operations were implemented through soft logic, which is very inefficient. Given the relevance of signal processing and telecom applications for FPGAs, FPGA architects devised new hardened components, namely DSP Blocks. DSPs, or hardened multipliers, were introduced by Xilinx Virtex-II [77]. An N-bit multiplier builds on N^2 logic elements and 2N inputs and outputs. DSP designers introduced new configurability support throughout the years, enabling the DSP fracturability and their tailoring for different applications and precision requirements. Additionally, vendors introduced input registers, adders, and output blocks to perform additional operations such as sum and accumulations. Overall, although DSP blocks were configurable, they natively support mainly fixed precision multiplication. Though telecommunication applications were (and still are) among the driving DSP factors, High Performance Computing (HPC) drove FPGA vendors to add native single-precision floating-point support to their DSPs. On a different direction, Deep Learning inference demands reduced precision computations, which strongly candidate FPGAs' datapath adaptability to this field. Researchers propose solutions to fracture DSP blocks to integer multiply further and accumulate operations at 9, 4, 2 bits. In this way, the DSP arithmetic requirements span from low-precision fixed-point/integer for inference, to medium fixed-point for telecommunication, to high-precision floating-point for HPC. It is up to the FPGA architect to design DSP blocks and eventually tailoring them to the specific application domain, e.g., Intel Stratix 10 NX [78], Speedster7t from Achronix [79].

2.1.6 System-level Interconnects, Interposers, and Others

The continuous growth of heterogeneity in the FPGA reconfigurable fabric goes hand in hand with the increasing external I/O interfaces bandwidth and capacity (e.g., DDR, PCIe). However, handling such a high volume and frequency traffic with soft fabric requires wider soft buses. For instance, a channel of High-Bandwidth Memory (HBM) is 128 bit wide and operates a 1GHz in a double data rate interface [64]. Thus, a soft bus that runs at 250 MHz must be 1028 bits wide. Given that the increasing number of high-speed interfaces can consume a massive portion of FPGA logic and routing fabric, new **hard system-level interconnects** became paramount. Indeed, latest Versal [80] and Speedster7t [81] embed a hard Network on Chip (NoC). The former employs a mesh-based NoC to connect the differ-

ent components such as I/O transceivers, CPU, programmable logic, and vector engines. The latter exploits the NoC as the primary communication among the external interfaces and the reconfigurable fabric. One of the logic growth has been the early adoption by FPGAs of **interposer** technology to enable dense interconnection of multiple silicon die at a reasonable cost [67,82,83]. A passive interposer is a silicon die with metal layers forming routing connections and several microbumps on its surface to connect more die flipped on top of it. This alternative technology achieves a higher yield compared to large monolithic single die systems. Moreover, interposer technology enables the integration of specialized chiplets with different technology nodes into a single device. The largest interposer-based FPGA has more than twice the logic elements of the largest monolithic-based FPGA (considering the same technology node). The next generation Xilinx Versal NoC further enhances this process connecting multiple die. Conversely, Intel-based FPGAs employ smaller interposers called Embedded Multi-die Interconnect Bridges (EMIB), which are carved into package substrate. They employ the EMIB to decouple design and technology nodes of smaller I/O transceiver or HBM chiplets.

Another crucial component of FPGAs is the **clock distribution network**. Indeed, FPGAs present special clock routing interconnections and on-chip clock generators (such as PLLs, Delay Locked Loops (DLLs), and CDR circuits) to offer programmable logic regions working under different clock domains. The clock routing networks employ a similar principle to those of Section 2.1.2. However, they aim at building low skew networks (e.g., H-trees) with reduced crosstalk and jitter effects. Last but not least, the FPGA **configuration circuitry** is responsible for configuring all the previously described programmable components. A configuration controller loads a defined bitstream on the device power-up. This circuitry can be exploited to enable the so-called *Dynamic Partial Reconfiguration (DPR)* of the soft logic. This functionality allows reconfiguring only a region of the programmable logic while running the rest of the circuit. Moreover, FPGA-based applications (i.e., the final bitstream) represent intellectual property, hence must be protected. Indeed, FPGA toolchains allow the creation of encrypted bitstreams that are decryptable only at runtime with many security implications [31].

2.2 Early Days Overview

Though RC provide a high level of flexibility and good performances [22], those systems require the body-of-knowledge from a wide mixture of fields

[25]. Therefore, the development of FPGA-based solutions was confined to few groups strongly specialized in hardware development due to both flaws in the development tools, the relatively low use of these fabrics, and the time-consuming process. Hence, the main users of RC systems were mainly those that developed system-wide competencies, from the high-level software down-to the single LUT [25]. Therefore, this gap among software and hardware productivity opens to a huge portion of research investments in CAD tools and the raising of abstraction levels for RC [84]. At the dawn of reconfigurable computing, HDLs were the first attempts to move from schematics to a higher productivity approach, but, to reach satisfactory results, the developer needs an in-depth knowledge of the underlying physical architecture of the fabric [25]. The required knowledge, indeed, prevents the accessibility to the field to a broader audience, for this reason, new environments were developed. AutoPilot, from 2013 Vivado HLS, succeeded in introducing the HLS, or programming the hardware with a high-level language such as C, in the RC industry [58]. Unfortunately, the efforts from the RC community were not enough, as demonstrated by a heterogeneous group of researchers [59]. The results of their analysis show that, with the current status of HLS tools, a software developer could not program FPGAs by simple software paradigms but must be aware of the considerable difference between the software optimizations and the hardware one [59].

While HLS represents an impressive step to increase the audience of reconfigurable systems, the computing world was shaken by the first version of the Project Catapult [32], a reconfigurable fabric in a custom data-center to accelerate large-scale software services, further extended in 2016 [33]. This example of success from a big corporation such as Microsoft broke the cliché of reconfigurable systems for a tiny niche. Nevertheless, the gap between hardware and software programming was still far from being filled up. Indeed, platforms such as GPUs have gained traction with the community and have been adopted as the standard way to accelerate computational workloads. The main reason is the ease of programming those devices, thanks to the abstraction offered by the Application Programming Interfaces (APIs) such as the Compute Unified Device Architecture (CUDA) [85]. The parallel computation power and the easy-to-use programming models have made GPUs the de-facto winner in several applications fields, such as machine learning model training.

2.3 Towards Reconfigurable System Democratization

The aforementioned huge productivity gap between reconfigurable systems and CPUs, pushed neither software programmers nor ASIC developers to embrace the RC world. Though the flexibility of reconfigurable systems opens to FPGA-based wireless sensor networks for tasks spanning from sensor fusion to small co-processor [56], the device complexity was increasing. Indeed, in 2011 the two main players in the FPGA market, i.e., Xilinx and Altera, introduced two heterogeneous systems composed of an FPGA tightly coupled with a hard-processor. Xilinx presents the Zynq technology [86], where an ARM processor and an FPGA are on the same chip, whereas Intel presents a Xeon coupled with an Altera FPGA [87] through the Intel QuickPath Interconnect (QPI) [88]. Additionally, technology advancements were on the roadmap for 2015 [89, 90], but the productivity gap, and the increasing device complexity, keep the reconfigurable devices for a niche. To increase their adoption, the research community has worked along two main lines of development. One line describes the democratization of reconfigurable systems with the improvement of CAD tools and the abstraction level provided to the final users, which move its first steps towards a domain specialization (Section 2.4.3). The second line focuses on a shift on how reconfigurable systems are employed, starting from Hardware-as-a-Service (HaaS), moving to increase specialization through heterogeneity, and finishing with DSAs.

2.3.1 Design Automation tools for FPGAs

One of the primary efforts in the RC world was centered on CAD tools for FPGAs. The increasing complexity of the available platforms, and the increasing demand for efficiency in computations, foster the research on CAD for RC systems. Many tools have been developed both at the academic and industrial levels. We will provide an overview of some of the most significant ones, clustering them into industrial and academic tools and closed and open sources.

The commercial tools are currently almost all closed source for obvious market reasons. Here we report some of the most relevant vendors on the market, such as Xilinx and Intel FPGAs, known as Altera up to 2015[4]. Starting from 2012, Xilinx released the Vivado design suite to support the latest released platforms [91]. Vivado, with Intel Quartus [92] as its counterpart, performs the system design task such as the synthesis, the place

[4] What follows it is not to be intended as, neither it is a complete commercial tools list.

and route, and the final bitstream generation, as shown in the left-hand side of Fig. 2.5. Both these companies provide their commercial version of HLS tool to enable fast prototyping and deployment, namely Vivado HLS [58] and Intel HLS Compiler [93]. Finally, both Intel FPGAs and Xilinx provide support for OpenCL language with the Altera FPGA SDK in 2012 [94, 95] and Xilinx SDAccel in 2014 [96], which has been unified in Xilinx Vitis [97]. One of the peculiarities of SDAccel, for example, is the complete abstraction from the underlying system, leaving the final user's only duty to develop the custom accelerator. This custom computing accelerator will be, in the end, integrated by the tool with a basic shell of logic blocks, thanks to a partial reconfiguration of the FPGA. On the other hand, the academic community continuously work to push further the research on CAD tools. One great effort has been put in HLS toolchains, reviewed in [59]. Among those, in the survey, there are two open-source solutions, such as BAMBU [98] and LegUp [99], and another one closed source named DWARV [100]. As mentioned in Section 2.2, the results show that, for a software developer, it is still discouraging to approach HLS tools. Indeed, newer alternative solutions continue to emerge, such the usage of custom Intermediate Representation (IR) [101], to push the development of Coarse Grain Reconfigurable Architectures (CGRAs) [102–105]. Furthermore, many research groups focus on the effective usage of the polyhedral model [106], widely used in software compilers, for the efficient automatic code generation targeting reconfigurable systems [107–110]. Other works try to overcome some limitations of many HLS tools, such as the expression of static parallelism and static scheduling [59]. An example comes from LegUp [111] that provides support to express multi-core hardware systems through multi-threading executions. Another body of work leverages existing toolchains to provide a testing environment for custom algorithms for the HLS phase, such as new architectural templates or new programming models, and for the design flow, such as new place and route algorithms. An example is the CAOS platform [112] that aims at increasing the adoption of RC systems in the HPC community, through a semi-automated hardware-software co-design flow [113] with a modular and extensible structure [114].

Following this trend, other open-source projects aimed at encouraging the development of custom algorithms within the FPGAs flow. Rapid Smith in 2011 proposed a set of tools and APIs for creating "your own custom CAD toolchain" on top of the so-called Xilinx Design Language (XDL) [115]. In 2018, Lavin and Kaviani presented RapidWright [116], a toolchain from Xilinx Research Labs. RapidWright is an open-source plat-

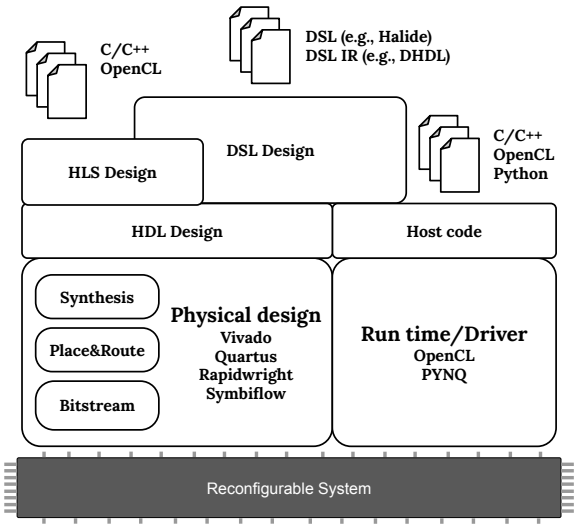


Figure 2.5: *Overview of the main blocks of design flows from source code down to reconfigurable system configuration*

form for custom module plug-in an FPGA-flow that aims to increase productivity and the design performance combined with the Vivado toolchain. Thanks to the proposed gateway to Vivado, called design checkpoints, the authors want to create an ecosystem around CAD tools for FPGAs [116]. On the wave of development of CAD for FPGAs, there is an interest in creating an ecosystem of open-source hardware tools. The Symbiflow project [117] aims at providing a fully free and completely open-source toolchain for commercial FPGA, with a flow from HDL down to bitstream generation, as in Fig. 2.5. Thanks to its first sub-project, named Ice-storm [118], they can reproduce a bitstream for Lattice iCE40 FPGAs, while currently, they are documenting the Xilinx 7-Series bitstreams [119]. In [120], the authors present their toolchain along with custom-computing machines, such as a low-power neural network and a Linux-bootable RISC System on Chip (SoC). Given the increasing complexity of reconfigurable platforms and the struggles related to the time-to-market, these open-source CAD tools can either improve the commercial tools with community contribution or democratizing reconfigurable systems.

2.3.2 The abstraction level rise towards Domain Specific Languages

Another important improvement of reconfigurable computing systems focuses on the rise of the abstraction level. Given the first CAD efforts that

Table 2.2: *Examples of research works in the abstraction level rise towards DSLs*

Language	Hardware design	Run-time mangament
Python		PYNQ [61]
OpenCL	Altera [121], Xilinx [122]	Altera [121], Xilinx [122]
Java	MaxJ [123]	Max run-time [123]
Scala	Chisel [124]	
Halide	FROST [125], Pu et al., [126]	Pu et al., [126]
Darkroom	Hegarty et al., [60]	

exploit C/C++-based HLS, the software community starts to approach the reconfigurable ecosystem while seeing an explosion of higher-level languages, such as Python, or to using more and more DSLs, such as Halide. Table 2.2 shows an overview of some of the relevant works in this trend. Indeed, a large amount of work aims at embodying high-level languages, different from C/C++, for new hardware-software co-design techniques, left-hand side Fig. 2.5, and run-time management, right-hand side Fig. 2.5, enabling a wider set of users.

For instance, the PYNQ project [61] is an open-source framework that enables Python programmers to use complex SoCs, or accelerator boards, supported by a set of predefined libraries and drivers. Another great effort by Altera comes from the integration of the OpenCL language in the FPGA-based design flow, first presented in 2011 [121]. In 2012 Altera released an official compilation framework for OpenCL-based designs along with a library for PCIe-based host-FPGA communication [94, 127] with encouraging results. Indeed, also Xilinx follows these efforts and provides integration in its HLS toolchain [122]. Both Altera and Xilinx exploit the versatility of the OpenCL standard for managing the run-time of the target platform and the hardware design, opening effectively to the idea of integrating PCIe-based reconfigurable accelerators in a server rack. A different approach is adopted by The Maxeler technologies, which provides integrated server-class CPUs with accelerators based on the dataflow model [128]. Moreover, they provide a design language called MaxJ, based on Java, which, compiled by the MaxCompiler tool [123], enables applications such as the development of design tools for CGRAs as in [101].

On the design side, several efforts have aimed at improving hardware-software co-design, although not all designed for RC systems.Chapter 3 will discuss more in detail these efforts, that refers to the RC stack in Figure 2.5. An example is Chisel [124], employed in the Edge Tensor Processing Unit (TPU) [129] or RC projects [130]. Among the DSLs, Halide [131] has been particularly attractive for the reconfigurable computing field, for

its focus on image processing (an exciting domain for RC) and for its ability to decouple execution and scheduling codes. Inspired from this language, several tools present framework for a DSL-to-FPGA design experience [60, 125, 126, 132, 133]

As highlighted in this Section, all these steps aim at opening the RC world to a broader public through tools that provide a more user-friendly use of these devices and pave the way for the first attempt of a paradigm shift, such as domain specialization. The following Section will guide the reader through the evolution towards the second line of development, namely the shift in the use of reconfigurable systems.

2.4 Recent Trends in the Reconfigurable Systems Spotlight

Aim of this Section is to provide an overview of the latest trends in the reconfigurable computing community. Specifically, reconfigurable computing systems have become one of the standard commodities available in the cloud and even more heterogeneous, with increasing problems linked to the communication infrastructure. Last but not least, there still is an open question on which is the most suitable use of reconfigurable systems. Should we tailor the reconfigurable system for a single domain with a wide range of applications, or should we exploit a single reconfigurable DSA engine, or is it better an automated tool to rule them all?

2.4.1 Reconfigurable computing in the cloud: Hardware-as-a-Service

Starting from 2014, a turning point changed the model of employing FPGAs in the cloud, when Microsoft [32] deployed Altera, and both IBM [34, 134, 135] and Baidu [35] deployed Xilinx FPGAs to improve their services. The idea of reconfigurable accelerators for cloud computing was gaining more and more traction, and in 2016, while Microsoft deployed a renewed and improved version of Catapult [33], with FPGAs attached directly to the network, one of the first surveys on this topic was published [55].

The year after the second version of Catapult, the terms of FaaS, or HaaS, become a steady reality. In 2017, Amazon presented the AWS F1 instances [36] with 1, 2, and 8 FPGAs devices attached, and Huawei presented its F1 instances of the FACS cloud service [37]. At that time, that meant that everyone could develop and deploy its FPGA-based service, system, and application without the need of owning the device per se and let a cloud provider manage and maintain the infrastructure. The FaaS revolution opens to new business models, new markets, and reality like Nimbix

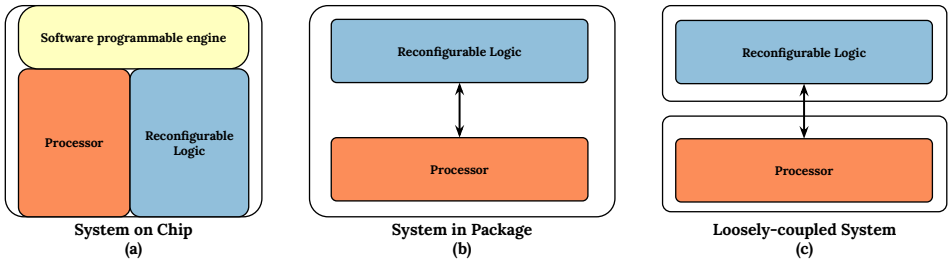


Figure 2.6: *Three classes of heterogeneous reconfigurable systems. While the first two, (a) and (b), devise a tight integration with/without a (private) shared memory, e.g., DRAM, the third one, (c) exploits (or not) memory coherency through interconnections or even directly attached to the network interface card.*

started to deploy its own HPC solution based on reconfigurable accelerators [136, 137], while it arises critical issues in the virtualization of reconfigurable fabrics [138]. Moreover, a project as FireSim [139] could rely on the FPGAs publicly available in the cloud. FireSim is an FPGA-accelerated hardware simulated environment that enables a more accurate representation of new data-center-like contexts to test either hardware or software design iterations without the need for a real deployment [140], and therefore limiting the cost for the final user.

2.4.2 Increasing heterogeneity in reconfigurable systems

Following the aforementioned improvements and considering that homogeneous multi-core processors, especially in data centers, fail to provide the desired energy efficiency and performance, new devices have been deployed, specifically heterogeneous architectures [54]. The integration in these architectures of hardware accelerators is gaining interest as a promising solution. Considering the different possibilities, FPGA-based heterogeneous devices apply to a wide range of fields thanks to their reconfigurability and high performance with low power consumption [141]. Based on these advantages, various platforms have been produced by the industry, with each of them employing different physical integration and memory coherency. Although these solutions are appealing, they pose different challenges to the developers, such as the choice of the most suitable one to a specific application [54]. As a result, we can identify not only different resolutive approaches for coupling CPU and FPGA, but also the integration of the FPGA with both CPU and software programmable accelerators, e.g., with Xilinx Versal ACAP [67], right-hand side Fig. 2.6. Looking at the pure CPU-FPGA coupling Choi et al., in [54], have provided an

exciting classification of platforms on the market. They have guided developers to decide which platform is most suited for a specific computational paradigm. For the scope of this work, we present some characterizing examples for the various approaches. Traditionally, the FPGA is connected to the CPU utilizing the PCIe interface with both of them with their private memories, such as Microsoft Catapult in its first version [32]. Other examples, which also allow the final user to use high-level languages for implementing its custom accelerator, are the Amazon F1 instances [36] and the Alpha Data FPGA boards [142]. These solutions enable the spread of reconfigurable fabrics as services to the final users and, allowing the use of high-level languages, open to a wider public. Also, based on PCIe interface, other vendors have proposed coherent shared memory between CPU and FPGA, such as IBM with its Coherent Accelerator Processor Interface (CAPI) for POWER8 [143]. Following the path of the coherent shared memory, but aiming at a tighter connection CPU-FPGA, the first version in 2011 of the Intel Xeon+FPGA platform exploits a QPI [87]. This idea evolved throughout the years, and in 2016 a further improvement of the Intel Xeon+FPGA platform was presented [144], center of Fig. 2.6. The version presented is a System in Package (SiP) where one, or more, reconfigurable accelerators are tightly coupled on the same package through the usage of a hybrid connection CPU-FPGA based on PCIe and QPI [144].

Though the improvements of the communication infrastructure were fostering the potential of these devices, as highlighted in [54], a coherent interconnection was still impractical because of the insufficient bandwidth and high latency cache designs. To this purpose, two consortiums of companies were born: the OpenCAPI group in 2016 [145], and the CCIX consortium in 2017 [146]. Their work produced two coherent communication standards, called OpenCAPI and CCIX, where the CCIX is compatible and built on top of the PCIe stack, while OpenCAPI is an open and new interface right-hand side of Fig. 2.6. Indeed, the OpenCAPI aims at being an open interface architecture that allows different ranges of accelerators, from smart NICs to reconfigurable-/ASIC-based systems, to be connected to the same high performance and coherent bus in an agnostic way concerning the processor architecture [147].

2.4.3 Towards domain-specialization

With the approach of physical technology limits, such as the end of Dennard Scaling [4] and slow down of Moore's law [8], computer architects have to face the growing computing demand differently. Unless a new dis-

rupting technology appears on the market, CPUs have reached a limit, and DSAs are the most viable way for energy-efficient computations [8].

In these regards, reconfigurable computing systems represent one of the possible solutions that enable custom-domain-specific computing platforms. Indeed, the literature presents many works that focus on a single domain, intended as a wide range of problems solvable with a common approach. However, the major problem now resides in an open question. Is it better to have a single DSA, a single computation engine to rule them all, or a tool able to generate an application-specific architecture for a target algorithm? Besides, the DSAs have different meanings in the reconfigurable computing world: is a DSA a "fixed" architecture and datapath that exploit the adaptability of a reconfigurable platform, or is it an architecture that is coarsely grained reconfigurable at the datapath/processing element level, more like a CGRA? Though CGRAs could impact remarkably, thanks to their low reconfiguration time compared to FPGAs, and their high specialization, "they are still immature in terms of programmability, productivity, and adaptability", as advocated in [38]. The paper presents a comprehensive and in-depth analysis of CGRA. To avoid misleading definitions, we refer to FPGAs as fine-grained reconfigurable architectures, or at the time of writing, devices available on the market. Instead, we refer to CGRA as reconfigurable computing platforms at a coarse- , or processing element-level [38]. Finally, we refer to reconfigurable processors, or Application-Specific Instruction-set Processor (ASIP), to a programmable CPU with custom logic, i.e., an ASIP, with a portion of reconfigurable logic devoted to implementing reconfigurable functional units [148, 149].

Among the several possible domains, Machine Learning (ML) has found its application in several fields, from computational theory to software engineering, from fraud detection to video segmentation, and many companies are born to address several problems within this domain. The reconfigurable computing world contributes to the growth of the machine learning world, especially considering Neural Network (NN), or Deep Neural Networks (DNNs), inference. Among the startups born from this context, DeePhi, now acquired by Xilinx, focuses its proposition on an efficient hardware-software co-design methodology to efficiently map NN-based computations on either an FPGA-based Deep Processing Unit (DPU) or an ASIC-based DPU [150]. Several works proposing a methodology to build efficient NN inference accelerators based on an FPGA have been proposed in the literature, and we suggest the reader interested in more details to look at [40]. Indeed, Guo et al., present a survey of efficient techniques to build NNs accelerators to focus on the design of the architecture, on how

to compress the model, and on how to design the system. A particular example not reported in the survey is the Neural Processing Unit (NPU) architecture of the Brainwave project [151], which aims at serving in real-time DNNs-based applications at the cloud-scale. A different example of a DSA for the ML world is described in [42]. The authors propose a methodology for stream-dataflow execution model based on a CGRA architecture organization. The reconfigurable datapath of the Softbrain microarchitecture has shown both comparable performance and energy efficiency results against state-of-the-art ASIC, whereas it keeps enough flexibility to reduce design and verification time, thus costs and time to market. Alongside all these considerations, Venieris et al., propose an extensive survey of toolchains for mapping Convolutional Neural Networks (CNNs) on FPGAs [41], that we suggest as reference for the topic. Specifically, the survey analyzes in-depth all the software-hardware automation tools used for CNNs mapping on FPGAs and proposes an interesting classification among the considered architectures. The target hardware of these toolchains can be divided into streaming architectures, that builds on top of a highly optimized basic block composed differently for each CNN, such as FINN [152], and single computation engines, a fixed architecture that generally varies software instruction sequences for different CNNs, such as FP-DNN [153]. Finally, both [40] and [41] conclude demanding an increased effort for hardware-software co-design tools in such domain.

Moving to a completely different domain, such as communication networks, where reconfigurable computing plays a crucial role, many works in the literature are torn between two approaches: a mapping DSL-to-hardware or a single DSA to rule them all. In these regards, a Reconfigurable Match Tables (RMT) architecture was proposed in 2013 [154]. Even if the authors tend to an ASIC-based DSA, the architecture proposed is a reconfigurable packet processing architecture that is software programmable. This work leads to the birth of a DSL in the field of switch architecture for packet processing that nowadays is widely recognized in the community and known as P4 [21]. As DSL, P4 is designed to be "reconfigurable", or software programmable, and protocol independent. Moreover, it abstracts completely from whatever specific packet format, and it is architecture-agnostic, meaning that the burden of targeting the underline architecture is left to the compiler. The first work that provides automation of generating HDL from high-level P4 programs is presented from Benacek et al., [155], and then expanded in further parallelism levels from Wang et al., [156] reaching outstanding throughput of $Tbits/s$ on an FPGA. On the other hand, Pontarelli et al., present their DSA, called FlowBlaze [157]. The ab-

Table 2.3: *Summary of different approaches to the domain-specialization of reconfigurable systems*

Domain\Approach	DSA	Tool	Hybrid
	"Fixed" architecture software programmable	One architecture for each problem to tackle	"Semi-fixed" architecture with a reconfigurable datapath
Machine Learning	NPU [151] FP-DNN [153] Deephi [150]	FINN [152]	Softbrain [42]
Networking	Flowblaze [157]	P4-to-VHDL [155, 156]	N/A
Regular Expressions	TiReX [48], CICERO [52]	REAPR [159], FlexAmata [160]	N/A

straction model they built upon is different from the one of P4, instead, it uses the same abstraction of the RMT architecture, called OpenFlow [158], and therefore can be considered as the RMT extension. Table 2.3 summarizes the division of the approaches adopted to domain-specialization, but Chapter 3 gives more insights on this topic.

2.5 Final Remarks

In this Chapter, we have summarized the evolution of the RC world, mainly focusing on FPGA-based systems and the concurrent development of new tools for helping the spreading of these technologies to recent days. In our journey in the reconfigurable fabric world, we have gone through different paradigms of use, and through the improvement of the support tools to developers, starting from the first attempts addressed to an already skilled audience till the more recent solutions that opens to a new user base, like software engineers. In these attempts, we have moved from solutions that can reach top performance only with an in-depth knowledge of the underlying system to the latest tools which guide the users during the entire process, from high-level code to the deployment of an entire system. In the wake of the opening to different audiences, reconfigurable fabrics have become a valuable option in data centers, given their reconfigurability, high performance, and lower power consumption than general-purpose multi-core architectures. On the other hand, thanks to their entry into the cloud market, a new paradigm that has become popular, is the concept of HaaS, where the final users exploit hardware resources of a cloud provider, which takes care of the maintenance and management costs of the physical board.

CHAPTER 3

On the Abstraction of Digital System Design: How to Program FPGAs

This Chapter describes the three digital abstractions on how to program Field Programmable Gate Arrays (FPGAs), spatial architectures with a heterogenous reconfigurable fabric. Since FPGA's adoption found limitations in programmability and required skills, many researchers are devoted to abstractions and automation tools. Therefore, for each of the considered abstractions (Hardware Description Language (HDL), High-Level Synthesis (HLS), and Domain-Specific Language (DSL)), this Chapter presents a taxonomy (programming models for HDLs; IP-based or System-based toolchains for HLS; application, architecture, and infrastructure domains for DSLs), and describes the prominent exponent with a timeline. Finally, it concludes with six relevant takeaways identified throughout this Chapter.

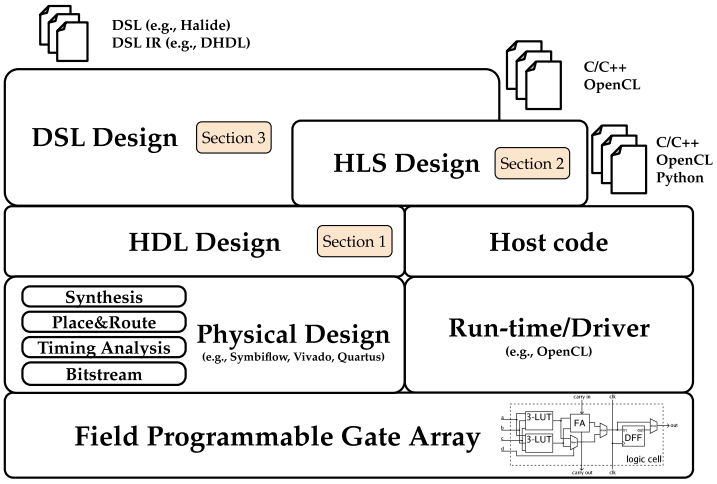


Figure 3.1: *FPGA Design Flow with reference Sections numbered*

As Section 2.1 introduced, Field Programmable Gate Arrays (FPGAs) are reconfigurable heterogeneous circuits able to implement custom digital systems, unlike fixed data paths and topologies Central Processing Units (CPUs). Their main usage was limited to prototyping and the telecommunication field, however, the internal complexity of FPGAs is incredibly growing [26, 64] Hence, academic researchers and companies started investing in FPGAs and adopting them also as accelerators [30, 32, 41, 161], providing a good trade-off between the flexibility of general-purpose CPUs and the performance and energy efficiency of Application-Specific Integrated Circuits (ASICs) [22–25]. Despite the great opportunities, the fundamental drawback of FPGAs has always been their challenging design process, profoundly impacting their programmability and steeping the learning curve. The hardware design flow for FPGAs resembles the one available for ASICs (Physical Design block of Figure 3.1). Historically, the primary way to develop hardware design for FPGAs and ASICs consisted of using Hardware Description Languages (HDLs), especially Verilog and VHDL, for standard Register Transfer Level (RTL) design. Indeed, many commercial Electronic Design Automation (EDA) tools [92, 162–164], and open-source tools [165], take RTL description in input and, from that, perform a sequence of steps towards the generation of the circuit. However, to efficiently leverage these languages, the user requires significant knowledge and experience in hardware design. Besides, despite the architectural evolution of FPGAs, the features and abstractions offered by Verilog and

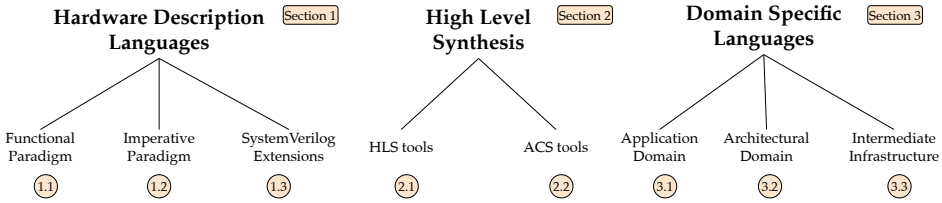


Figure 3.2: *Proposed taxonomy for the considered digital abstractions*

VHDL did not evolve as fast. Thus, over the last years, new solutions have emerged to cope with the current limitations. In this scenario, we can identify three main categories of novel digital design abstractions: high-level HDLs, High-Level Synthesis (HLS) tools, and Domain-Specific Languages (DSLs), as the numbered blocks in Figure 3.1. Modern HDLs offer features not available in traditional HDL, while providing a design experience close to the hardware. HLS tools enable designers to rely on high-level languages to design hardware architectures. Finally, DSL tools represent newer trends to increase further productivity, performance, and ecosystem exploration thanks to domain narrowing. Independently from the category, the goal of such tools is to: increase the level of abstraction and productivity for hardware design; enable high reuse and customization of IPs; reduce verification effort and design errors; make FPGAs accessible to a broader audience of users. This Chapter describes the research efforts on digital design abstractions for FPGA programming divided into *HDLs* (Section 3.1), *HLS* (Section 3.2), and *DSLs* (Section 3.3). For each of the three abstraction efforts, we consider the review few examples of active languages and tools[1]. Besides, we provide: a **timeline** (Figures 3.3, 3.5 and 3.6), reporting the first available dates of the tool appearance; a **taxonomy**, reported in Figure 3.2, based on trends that we identified, such as Programming Model (Figure 3.3), Synthesis Target (Figure 3.4) or Target Domain (Figure 3.6); a **review of the main characteristics** of each analyzed work (Tables 3.1 to 3.3); insights on possible future trends (Section 3.4).

3.1 Hardware Description Languages

An HDL aims at describing the behavior of digital logic circuit designs both for ASICs and FPGAs[2]. In the 1980s, VHDL and Verilog emerged as HDLs introduced to help the electronic designers in the simulation and verification

[1] Some of the excluded are available at [17] [2] Excluding analog and mixed analog-digital circuit design.

of Integrated Circuits (ICs), still needing the human for HDL to schematic translation [65, 166–168]. With the advent of logic synthesis and digital circuits' growth, EDA vendors pushed HDLs from just simulation and verification languages to design languages. However, for the most prominent HDLs from the 1980s (i.e., Verilog and VHDL), a significant portion of the language is not thought for synthesizing the circuit itself but for simulation purposes. Indeed, we speak of *synthesizable* or *non-synthesizable* constructs when speaking of VHDL and Verilog [65, 166–168]. Currently, VHDL and Verilog, now merged in SystemVerilog, are IEEE standards, part of commercial EDAs tools, and the de facto standard for many FPGA and ASIC designers, though not the only alternative. The impressive technology improvements increased the complexity of the hardware devices (e.g., in terms of logic gates and heterogeneity [26, 64, 65] as highlighted in Figure 2.2) demanding continuous research on tools able to handle such complexity and support the designers. Meanwhile, both academic and industrial users found limitations in the two standard HDLs and their productivity, hence proposing new programming paradigms. Some of these HDLs have their own syntax and constructs, while others are embedded in high-level languages like Scala and Python. Eventually, each of these HDLs translates into VHDL or (System)Verilog[3], as they remain the only languages supported by modern synthesis tools.

We propose an HDL taxonomy based on the characteristics of the programming model employed in the input languages. Our taxonomy has three main clusters: HDLs based on functional languages (Section 3.1.1), HDLs based on imperative languages (Section 3.1.2), and SystemVerilog extensions (Section 3.1.3). Figure 3.3 shows the taxonomy of modern HDLs, their year of birth, and their input and output languages.

3.1.1 Functional-based HDL

The first category of HDLs derives from functional programming languages and embeds the characteristics of such a paradigm within the low-level hardware development flow. There are two different languages employed by HDL developers: Haskell [169] and Scala [170]. While the first language is mainly devoted to functional features, Scala provides abstractions for object-oriented style, making it very appealing for new languages [124] and represents the current wave of functional-based HDLs. The great variety of abstractions and support from a growing community are essential factors that continuously push the development of these languages to embed low-level design features of native HDLs such as multi-clock domains.

[3] This means both Verilog and SystemVerilog

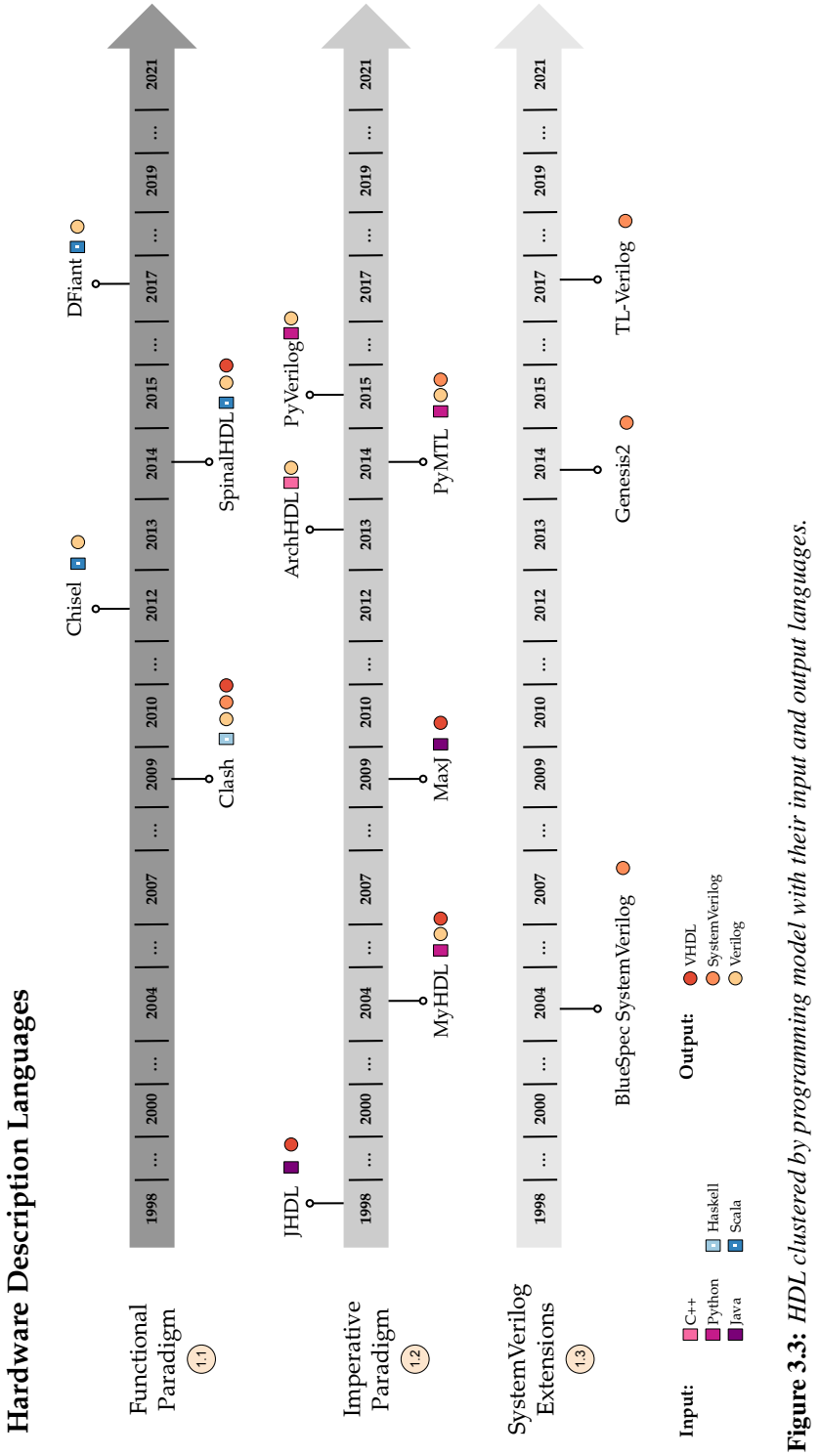


Figure 3.3: *HDL clustered by programming model with their input and output languages.*

Chisel: Chisel [124] is an HDL embedded in Scala, which offers a more straightforward approach to HDL design compared to Verilog. For instance, the designer can define functions using Scala conventions, build and nest data structures, design components as classes, and redefine operators. Chisel specific libraries permit the designer also to employ custom data types. A key for embedding Chisel in Scala is to support highly parametrized circuits generators, a weakness of traditional HDLs. In this way, designers can declare parameterizable classes and recursively create hardware subsystems. For instance, RocketChip System-on-Chip Generator [171] is a Chisel-based framework. As another exciting 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 and a Verilog translator, which permits fine changes and integration with already designed Verilog modules as black-boxes. Additionally, Chisel supports multi-clock domain designs, and it is used for a plethora of FPGA and ASIC designs [171–173], though it lacks in verification features [172].

3.1.2 Imperative HDL

The second category of HDLs relies on imperative paradigms, such as those offered by Java and Python. These languages' programming model is closely related to hardware description where components are reused across hierarchical and, whenever possible, decoupled designs. Although Scala-based HDLs adopt object-oriented features, they provide many functional constructs that collide with this category. This class of HDLs mainly exploit objects and classes along with polymorphism features. These features are at the basis of an easy-to-use and extend language [174–176] and tight integration with the target host machine [177, 178].

MyHDL: *MyHDL* [174] is a language that exploits the Python infrastructure to implement HDL specifications to open hardware development to beginners. Its HDL description is similar to Verilog, but with a more manageable approach to verification; indeed, it is possible to convert MyHDL code into Verilog/VHDL through specific built-in libraries and use constructs to verify the designs easily. MyHDL supports waveform viewing as well. MyHDL models hardware as interactive light-weight threads that communicate with each other. In particular, MyHDL description structure is based around *generators*, namely modules that wait for a specific signal to perform specific actions, that communicate through *generator functions*. Moreover, generator functions allow to keep the state of the employed functions and resume them if needed, making them usable as ultra-light threads.

In this way, it is possible to pass control information to the dedicated runtime simulator. Finally, MyHDL supports co-simulation via other HDL simulators by translating MyHDL code into Verilog.

3.1.3 SystemVerilog Extension HDL

First appeared in 2002 [179], SystemVerilog represents a significant improvement over his predecessor Verilog. Thanks to many features borrowed from the object-oriented world, SystemVerilog increases the abstractions for hardware developers (both design and verification people). The third category of HDLs based its power on extending SystemVerilog syntax for different purposes: from a new design experience to highly customizable hardware generators [180], and new design paradigms, such as the transaction-level paradigm.

BlueSpec SystemVerilog: BlueSpec SystemVerilog (BSV) [181, 182] is an HDL that aims to provide a general-purpose language for hardware design using *atomic transactions* to deliver concurrent execution and easy reconfigurability. Atomic transactions are rules that dictate the behavior of the described hardware to enable a high level of parallelism and smoothly refinable designs. The designer develops modules in BSV and implements, for each module, both methods and rules. The modules represent the outwards interfaces, while the rules update and modify the module's internal state. Both rules and methods have guards, and they can fire only if the guards are true and there are no conflicts concerning the considered rule, preserving atomicity. The designer can change the application order of the rules without modifying the rules themselves, differently from SystemVerilog. The BSV synthesis tool compiles parallel hardware for the rules, which is always logically equivalent to a serialized execution. Module interfaces are components of atomic transactions and derive from C++ and Haskell interfaces. BSV permits polymorphism to easily create complex, overloaded, and fully type-checked interfaces in a bottom-up approach by constructing templates. Designers can easily design reusable components to build a more complex architecture. Indeed, the generation mechanism of micro-architectures supports conditionals, parametrization, loops, and even recursion, making the design process more comfortable and more customizable. Moreover, BSV modules can coexist with SystemVerilog blocks, thus giving the developer the possibility to use already existing ones. Finally, BSV supports design with positive clock edge and reset asserted low by default, multiple clock domains, and DDR designs, i.e., logic active on both positive and negative clock edges. Despite this possibility, BSV does not

support timing verification, which requires a standard Verilog simulator.

3.1.4 Summary

Table 3.1 summarizes the presented HDLs, clustered by programming model (functional, imperative, SystemVerilog-based) and then ordered by date of appearance. The first two HDLs clusters present many languages in contrast to the reduced number of SystemVerilog extensions. Functional and imperative clusters leverage high-level languages constructs and formalisms to enhance the language expressivity to design parallel hardware architectures. Conversely, the third cluster, which is smaller, extends an HDL such as SystemVerilog through language features that further ease the hardware development, e.g., BSV rules or TL-Verilog transactions.

All three clusters contain open-source languages, and their majority reports updates in the last two years (2020-2021). This fact introduces a community building around languages that come even before 2010 (e.g., Clash) and interests beyond the single research manuscript. Each of these languages outputs standard HDL language, either VHDL or (System)Verilog. However, the majority of the considered HDLs exploit only one of the two standards as the output language, especially Verilog. Indeed, only Clash, SpinalHDL, and MyHDL support both VHDL and Verilog outputs.

As HDL relevant characteristics, we highlight if the target language supports parametrization and polymorphism, two essential features for abstraction improvements. Moreover, Table 3.1 displays the simulation support (if any) with a custom tool or if they leverage third-party tools, and other features we believe relevant in the last column. However, without some of the traditional HDLs features these languages are not considered ready for production purposes [172]. Finally, among the presented HDLs, we see a promising direction from novel languages, such as TL-Verilog [190]. However, we believe that the most promising and mature languages are the ones that provide standard HDL features (e.g., BlackBox IP, Verification), additional ones (e.g., higher abstraction constructs), a continuous development throughout the years, and a consistent ecosystem. Examples are Chisel, SpinalHDL [184], and BSV.

3.2 High-Level Synthesis

After discussing relevant HDLs in State of the Art, we now focus on HLS tools [191, 192], which increase the abstraction level of digital design flow. HLS aims to enabling users, not necessarily expert in the hardware domain, to develop a digital custom architecture for FPGAs or ASICs starting from

Table 3.1: *Comparison table of the presented HDLs*

	HDL Language	Input Language	Output Language	Parametrization	Polymorphism	Simulation	Other Relevant Features
Functional	Clash [183]†‡⋆	Haskell	VHDL, (System)Verilog	Supported	Supported	Supported	Multi Clock and Reset Polarity
	Chisel [124]†‡⋆	Scala	Verilog	Supported	Supported	Supported	Multi Clock Design, BlackBox IPs, functional Verification
	SpinalHDL [184]†‡⋆	Scala	VHDL/Verilog	Supported	Supported	Supported	Multi Clock designs, DDR designs, BlackBox IPs, timing Verification
	DFiant [185]†	Scala	Verilog	Supported	Supported	Supported	Dataflow, Clock-less
	VeriScala [186]†	Scala	Verilog	Supported	Supported	Supported	Recursion, Scala-FPGA run-time
Imperative	JHDL [177]	Java	VHDL	Supported	Not Supported	Supported	Host Design, Partial Reconfiguration
	MyHDL [174]†‡⋆	Python	VHDL/Verilog	Supported	Not Supported	Supported	Verification
	MaxJ [178, 187]⋆	Java	VHDL	Supported	Supported	Supported	Host Design
	ArchHDL [188]	C++	Verilog	Not Supported	Not Supported	Supported	-
	PyMTL [176]†‡⋆	Python	(System)Verilog	Supported	Supported	Supported	-
	PyVerilog [189]†‡⋆	Python	Verilog	Not Supported	Not Supported	Not Supported	-
	PyRTL [175]†‡⋆	Python	Verilog	Supported	Supported	Supported	Instrumentation
SystemVerilog Extensions	BlueSpec [181, 182]†‡⋆	Extension of SystemVerilog	SystemVerilog	Supported	Supported	Supported	Multi Clock and DDR designs, functional Verification
	SystemVerilog †	Extension of SystemVerilog	SystemVerilog	Supported	Not Supported	Supported	-
	TL-Verilog [190]⋆	Extension of SystemVerilog	SystemVerilog	Not Supported	Supported	Supported	Timing Verification

† Open-source ‡ Last update in the last years (2020-2021) ⋆ Maintained, to the best of authors' knowledge

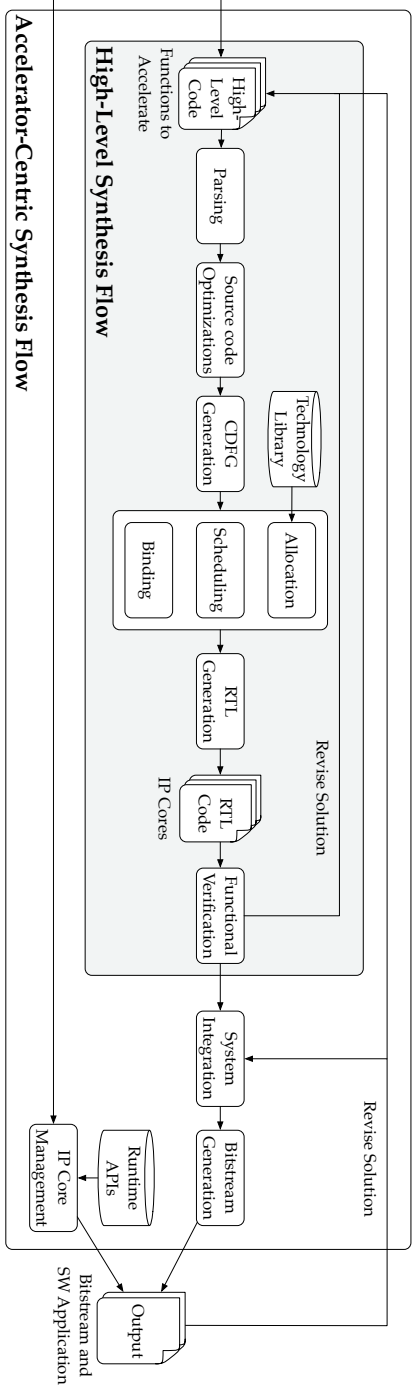


Figure 3.4: *High-Level Synthesis and Accelerator-Centric Synthesis flows and their integration*

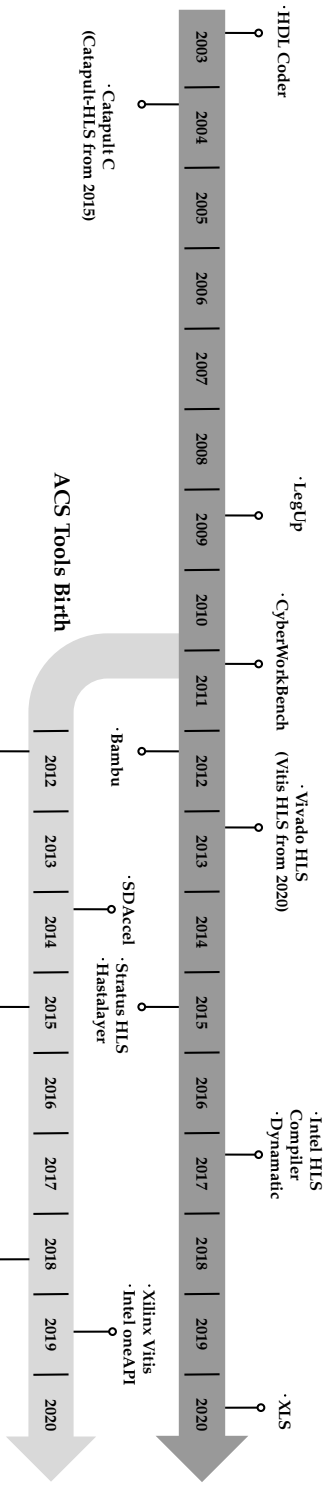


Figure 3.5: *Timeline of High-Level and Accelerator-Centric Synthesis Tools (first release/published paper).*

a high-level language. Given an algorithmic/functional specification (untimed description) of a system (often decorated with directives/pragmas), the HLS tool translates it into an intermediate representation (usually a control and data flow graph). From this point, the HLS tool determines the types of operators and memory elements the specification needs and *allocates the resources*. Then, the next step *schedules* the operations within the specification to clock cycles. Later on, the tool *binds* each operation and variable to a specific functional unit and a memory element, respectively, and circuit interfaces (control and data signals) to peripherals (such as memory interfaces). Finally, the HLS tool *generates a fully timed RTL design*. Thanks to this approach, HLS improves the design productivity and facilitates the exploration of the design space through source code and directives tweaking. Besides, HLS tools may reduce the verification time with automatic testbench generation. Despite the advantages, an HLS-based design flow has some relevant flaws. First of all, the user has less control over the resulting RTL design [193], significantly depending on the tool internals and optimizations, Moreover, the majority of HLS tools offer little to no support for specific kinds of RTL designs, e.g., cross-clocking domains, and it mainly focuses on datapath applications [194]; thus, it may not be the best choice for control ones. According to Martin et al. [194], we are currently in the third generation of HLS tools that leverage the two previous generation failures (e.g., immaturity of tools, improper input languages). For instance, most HLS tool vendors employs familiar high-level languages, like C variants (C/C++/SystemC) instead of specific languages. Thanks to their features, modern HLS tools boosted FPGA programmability and helped reduce the steepness of the learning curve, especially when the user's goal is the acceleration of a given application.

The efficient design of an accelerator is only a part of the whole FPGA design process. Indeed, the next step, usually called *system-level design*, involves integrating the resulting IP within a more extensive system. Therefore, the proper connection of the produced IP with on-board components is paramount to deploy the accelerator, allowing the user to interact with the IP from the host processor However, HLS tools focus on designing an IP and do not cover/automatize the system-level design step, which is usually up to the user. For this reason, in recent years, FPGA vendors started developing toolchains embedding both the HLS step and automatic system-level integration of the resulting IP oriented to the hardware accelerator design. We name such toolchains *Accelerator-Centric Synthesis (ACS)* tools: they represent a further step in the FPGA development panorama and offer a CPU/GPU-like development experience. We categorize them as

the fourth and latest generation of HLS, extending the previous classification [194]. Figure 3.4 depicts the tight integration between HLS and ACS flows. Throughout this Section, our analysis focuses on the most relevant HLS tools in literature, whether they just perform the RTL synthesis step (Section 3.2.1) or the entire flow towards the bitstream generation (Section 3.2.2), excluding toolchains that leverage such tools [195–197].

3.2.1 High-Level Synthesis Tools

This Section describes the current status of the most relevant third generation HLS tools ordered by date of appearance.

Vivado/Vitis HLS: Vivado HLS [198], formerly AutoPilot by AutoESL and then acquired by Xilinx in 2011. Vivado HLS accepts C, C++, or SystemC as input specification languages and can generate Verilog or VHDL hardware descriptions. The designer can specify the target FPGA and provide constraints on the clock period, clock uncertainty and optimization directives to better control the HLS process. Vivado HLS accepts most of the constructs of C/C++ while applying the usual restrictions, such as recursion and dynamic memory allocation. The designer can leverage multiple directives to improve the final design, such as loop transformations, binding to specific resources, hardware interfaces definition, and dataflow execution model. Vivado HLS supplies tools for functional verification of the resulting design at both software and hardware level with the same software testbench along with various reports on timing, usage, and scheduling. Finally, thanks to the integration with the other Xilinx tools, the designer can invoke synthesis and place & route steps within Vivado HLS to assess the design quality. From the 2020 release of its developer tools, Xilinx substituted Vivado HLS with Vitis HLS [199], which automatically applies more optimizations and relies on AMBA AXI4 interface protocol to communicate with the off-chip memory by default.

Dynamatic: Dynamatic [200] is an academic open-source HLS framework [201].This tool converts C/C++ code into synchronous dataflow circuits. The main feature of Dynamatic is its ability to schedule the resulting circuit dynamically. Usually, most HLS tools tend to schedule the output circuit statically, forcing worst-case assumptions and, consequently, reaching suboptimal results. Conversely, Dynamatic's approach supports the design of dataflow circuits able to adjust the schedule at runtime. Besides, Dynamatic leverages performance modeling to optimize the throughput through optimal buffers placement and sizing. Finally, thanks to its open-source nature, developers can extend Dynamatic by adding custom

features, pragmas, and optimization passes.

3.2.2 Accelerator-Centric Synthesis Tools

This Section analyzes ACS tools in chronological order, as depicted in Figure 3.5. These tools provide a unified environment where the user can design both the host and accelerator code and integrate them via high-level APIs. The ACS tool takes care of both the HLS process and the system-level integration according to the target scenario, i.e., host-to-IP communication via shared memory (embedded) or PCIe (high-end/cloud).

Intel FPGA SDK for OpenCL: Intel FPGA SDK for OpenCL [202], previously known as *Altera SDK for OpenCL* (AOCL) [94], is a development environment that enables software developers to accelerate their applications targeting heterogeneous platforms with Intel CPUs and Intel FPGAs. The designer develops both the kernel and host code within the same environment on top of the OpenCL computational paradigm. The tool inserts performance counters in the FPGA design, and the result obtained can then be reviewed by the designer using the Dynamic Profiler tool. Moreover, it provides analysis on the resources and performance, a fast FPGA-based emulation, *what-if kernel* performance analysis, and support for symbolic debugging. Once the host application and the kernel match the expected performance, Intel FPGA SDK for OpenCL performs a complete compilation towards the bitstream.

TAPAS: Built on top of a parallel Intermediate Representation (IR) called Tapir [203], TAPAS is a three stage open-source toolchain from Simon Fraser University [130]. The first one analyzes the IR and extracts task dependencies and the top-level Chisel (Section 3.1.1) module that instantiates the DRAM interface, a shared L1 cache coherent with the L3 cache of the core processor, and the task units. Then, stage 2 analyzes the program graph for each task and generates each unit's RTL dataflow. Finally, stage 3 configures a set of parametric components and generates the bitstream.

Vitis Unified Software: Vitis [204] is a unified software released by Xilinx in 2019, which incorporates the functionalities of previous tools like SDAccel (for datacenters) [205] and SDSoC (for embedded systems) [206], adding new ones. In particular, Vitis supports the development of accelerated applications and embedded software, targeting both high-end and embedded FPGA-based platforms. Also, Vitis relies on Vitis HLS [199] to synthesize C/C++ and OpenCL code into RTL. One of the key components available within Vitis is Vitis AI. This development environment permits users to accelerate the inference of Artificial Intelligence models on FPGA.

Starting from a high-level description in TensorFlow, PyTorch, or Caffe, Vitis AI optimizes and compiles the model into a binary for Xilinx's Deep Learning Processing Units (DPUs). Another relevant features are the set of open-source out-of-the-box accelerated libraries [207] and the improved integration with the Xilinx Runtime library (XRT) [208] for host-accelerator communications.

3.2.3 Summary

Table 3.2 reports the key characteristics of the analyzed HLS/ACS tools. Both companies and academia contributed to the growth and success of the third generation of HLS. In particular, a significant initial effort came from academia (Figure 3.5), whose research products, like LegUp, AutoPilot (now Vivado HLS), turned into commercial tools. Another pivotal factor that made this generation successful is the adoption of C-based languages. Indeed, most tools rely on C-based languages with specific restrictions, e.g., recursion, dynamic memory allocation.

Moving to the output of the HLS process, multiple tools generate RTL code suitable for more than one FPGA vendor, as well as ASICs. On the other hand, most ACS tools target just one vendor, which is also their developer. Indeed, since ACS tools perform both HLS and system-level integration steps, it is easier for an FPGA vendor to integrate different products within a unified environment and provide designers with a complete toolchain.

Considering other relevant aspects, we believe that functional verification of a hardware design is one of the essential features HLS tools should implement since it enables checking the correctness of the produced RTL code. However, various tools, especially "pure" HLS ones, offer partial functionalities for design verification, as they just supply one type of simulation (SW or HW). In contrast, most ACS tools provide a more comprehensive experience, implementing SW and HW simulation and even performance analysis via profiling tools. Similarly, modern ACS/HLS tools are now general and mature enough to enable the hardware design of applications belonging to different domains. Nonetheless, domain specialization has its advantages, and Section 3.3 analyzes research efforts in that direction.

In the future, we foresee that vendors will keep furthering their toolchains, especially ACS ones, to make FPGAs more appealing to software developers. Indeed, these tools, which we consider the fourth generation of HLS [194], resemble the software development experience more than the

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

	Tool	License	Owner	Input Language	RTL Language	Target Vendor	TestBench Generation	Simulation	Domain	Precision FP	Precision FixP	Other Relevant Features
HLS	HDL Coder [209]*	Commercial	Mathworks	MATLAB	VHDL, Verilog	Intel, Xilinx	Yes	SW, HW	DSP, Image Proc.	No	Yes	ASIC support, graphical design
HLS	Catapult-HLS [210, 211]*	Commercial	Mentor Graphics (now Siemens)	C++, SystemC	VHDL, Verilog	Intel, Xilinx	Yes	SW, HW	All	Yes	Yes	ASIC support
HLS	Kiwi [212]†	Academic	University of Cambridge	C#	Verilog	Intel, Xilinx	No	SW	Scientific	Yes	No	Multi-clock, recursion, dynamic allocation
HLS	GAUT [213]†	Academic	U. Bretagne Sud	C, C++	VHDL, SystemC	Intel, Xilinx	Yes	HW	DSP	No	Yes	ASIC support
HLS	ROCCC [214]†	Commercial	Jacquard Computing	C subset	VHDL	Xilinx	Yes	HW	Streaming	Yes	No	Smart buffers
HLS	LegUp [99]*	Commercial	LegUp Computing (now Microchip)	C, C++	Verilog	Microchip	Yes	SW, HW	All	Yes	Yes	Pthread and OpenMP support
HLS	CyberWorkBench [215]	Commercial	NEC	C, C++, SystemC	VHDL, Verilog	Intel, Xilinx	Yes	SW, HW	All	Yes	Yes	Multi-clock design, clock-gating, ASIC support
HLS	Bambu [98]†‡*	Academic	PoliMI	C	VHDL, Verilog	Intel, Xilinx, Lattice	Yes	SW, HW	All	Yes	No	Modular and extensible, ASIC support
HLS	DWARV [216]	Academic	TU Delft	C	VHDL	Xilinx	Yes	HW	All	Yes	Yes	Modular and extensible, external IP integration
HLS	Vivado/Vitis HLS [198, 199]*	Commercial	Xilinx	C, C++, OpenCL SystemC	VHDL, Verilog, SystemC	Xilinx	Yes	SW, HW	All	Yes	Yes	-
HLS	Stratus HLS [217, 218]*	Commercial	Cadence	C, C++, SystemC	VHDL, Verilog, SystemC	Agnostic	Yes	SW, HW	All	Yes	Yes	ASIC and SoC support
HLS	Hastlayer [219]†‡*	Commercial	Lombiq Technologies	.NET	VHDL	Intel, Xilinx	No	SW	All	No	Yes	Support for .NET framework languages
HLS	Intel HLS Compiler [220]*	Commercial	Intel	C, C++	Verilog	Intel	Yes	SW, HW	All	Yes	Yes	-
HLS	Dynamatic [200]†‡*	Academic	EPFL	C/C++	VHDL	Xilinx	Yes	HW	All	Yes	N/A	Dynamically scheduled circuits
HLS	XLS [221]†‡*	Commercial	Google	DSLX, C++	SystemVerilog	Agnostic	No	SW, HW	All	Yes	Yes	ASIC support, fuzzer, HW-oriented IR
ACS	Intel FPGA SDK [94, 202]* for OpenCL	Commercial	Intel	OpenCL	VHDL, (System)Verilog	Intel	Yes	SW, HW	All	Yes	Yes	System Integration Bitstream Generation
ACS	SDAccel [205]	Commercial	Xilinx	C, C++, OpenCL	VHDL, (System)Verilog	Xilinx	Yes	SW, HW	All	Yes	Yes	System Integration Bitstream Generation
ACS	SDSoC [206]	Commercial	Xilinx	C, C++, OpenCL	VHDL, Verilog	Xilinx	Yes	SW, HW	All	Yes	Yes	System Integration Bitstream Generation
ACS	TAPAS [130]†	Academic	Simon Fraser	Cilk(-P), OpenMP	Chisel	Intel SoC	No	HW	All	Yes	No	System Integration Bitstream Generation
ACS	Vitis Unified Software [204]*	Commercial	Xilinx	C, C++, OpenCL	VHDL, Verilog	Xilinx	Yes	SW, HW	All	Yes	Yes	System Integration Bitstream Generation
ACS	Intel oneAPI Toolkits [222]*	Commercial	Intel	C, C++, CUDA, OpenCL	VHDL, Verilog	Intel	No	SW, HW	All	Yes	Yes	System Integration Bitstream Generation

† Open-source ‡ Last update in the last years (2020-2021) * Maintained, to the best of authors' knowledge

hardware one, hiding and automating low-level technicalities like system-level integration. Besides, they represent the ideal entry point to both accelerating and deploying compute-intensive workloads vendors' accelerator cards. However, if, on the one hand, this approach helps the designer as it offers automated system-level integration and runtime APIs to manage the interaction with the FPGA, on the other, it constraints the development to a specific flow. For this reason, we believe that "pure" HLS tools, i.e., the third generation, will continue to exist as they enable fast software-like development of IPs for custom systems out of the scope of ACS tools.

3.3 Domain-Specific Languages for FPGA Design

Although they have been around for several decades [223], DSLs, and domain specialization in general, gained a lot of popularity in recent years [8, 11, 38, 42, 50, 224] for many reasons. First of all, modern DSLs enable developers to quickly and easily develop portable code for multiple architectures, especially CPUs and GPUs, increasing productivity. Then, the domain restriction allows DSL compilers to explore the design space quickly, identify the typical computational patterns of the target domain, and produce highly optimized implementations with unnecessary constructs [225]. Consequently, DSL applications may reach remarkable performance with a relatively minor design effort than other more general languages and frequently outtake hand-tuned libraries [131, 133].

Similar to CPUs and GPUs, DSLs are particularly convenient also for FPGAs. Thanks to domain specialization, the compiler can quickly explore the design space and leverage the FPGA features. In this way, the compiler relieves the burden of manually exploring various solutions, permitting designers to just focus on the functional description of the target algorithm and further reducing the learning curve steepness. This aspect is highly beneficial in the FPGA scenario, where the development of new solutions is highly time-consuming.

In this Section, we describe the most relevant DSLs targeting FPGAs available in the literature. Our analysis only examines languages of limited expressiveness developed for a specific domain or intermediate frameworks upon which such languages can build to target FPGAs. Therefore, we exclude other domain-specific tools that do not directly involve the usage of a DSL. For instance, although they perform a similar task, we exclude the various machine learning frameworks available in literature as they require as input a high-level description in a given format (e.g., JSON, Protocol Buffers) that does not fall under the DSL definition [223]. However, an in-

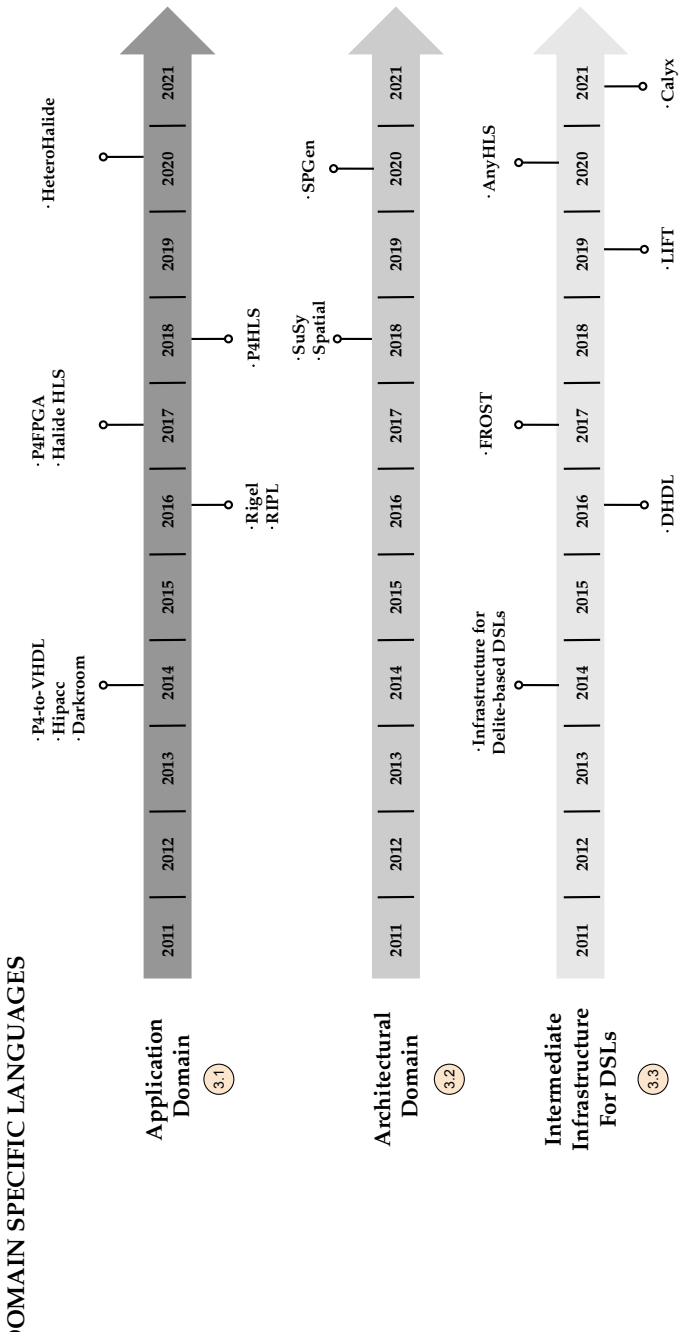


Figure 3.6: *Timeline of the reported DSLs and Intermediate Infrastructures from when FPGA support started.*

terested reader may look at other specific literature surveys for more details about such tools [41].

We group the DSLs in three main clusters according to a taxonomy based on the DSL features and purpose. The first cluster considers DSLs that focus on a particular application domain (Section 3.3.1), while the second contains the ones tailored to a specific architectural model (Section 3.3.2). Finally, the third cluster comprises intermediate languages and infrastructures for DSLs (Section 3.3.3).

3.3.1 Application Domain

This Section examines DSLs focusing on a given application domain. Such languages implement specific constructs and abstractions that ease the development of efficient code for image processing, packet processing, numerical solvers, and so on.

P4 Frameworks: P4 [21] is a high-level language for programming packet processors born in 2014 to answer the increasing demand for adaptable switches. Thanks to the wide employment of FPGAs in the networking field, many P4-to-FPGA frameworks were born after its release.

P4-to-VHDL is an experimental framework developed in 2016 that, starting from a P4 program, produces a VHDL-based architecture for packet parser at 100 Gbps [155]. Based on this work, Cabal et al. propose a new version of the target packet parser architecture that achieved even better throughput [226]. P4FPGA is a framework that provides a P4-to-BSV (Section 3.1.3) translation [156]. P4FPGA starts by taking standard P4 IR [21], performs an IR-to-IR transformation, then composes the basic blocks of the programmable pipeline, and finally emits the pipeline as BSV code for standard FPGA flow. Besides, P4FPGA produces a runtime system that provides hardware-independent abstractions for functionalities such as transceiver management, and host/control plane communication.

Darkroom: Darkroom [60] is an image processing DSL and a compiler embedded in the Terra language [227]. Designers can exploit Darkroom to realize image processing pipelines for FPGAs, ASICs, and CPUs. Darkroom expresses image processing algorithms as direct acyclic graphs of image operations, restricting them to fixed-size stencils. In particular, Darkroom implements such pipelines based on the *line-buffering* architectural pattern, which consists of storing intermediate data between the pipeline stages This pattern permits minimizing the memory bandwidth and improving performance and power efficiency. Given an input pipeline, Darkroom finds the minimal buffer size via an integer linear programming formulation

and automatically schedules the computation.

3.3.2 Architectural Domain

The second cluster of DSLs shifts the focus from the application level to the architectural one. Indeed, they implement a particular architectural model/template and offer features and constructs to support classes of algorithms that benefit from it. Eventually, the compiler takes the input code and customizes the underlying architecture to better fit it.

Spatial: Spatial [228, 229] is an open-source DSL and compiler for the design of spatial accelerators targeting FPGAs, Coarse Grain Reconfigurable Architectures (CGRAs), and ASICs. Spatial compiles the code to C++ (host) and Chisel (accelerator) through target-agnostic abstraction. According to the target device, Spatial relies on either Xilinx and Intel's toolchains, Plasticine CGRA [102], or Synopsys tools. Spatial builds upon four criteria that the developers considered necessary to offer a good balance between productivity and performance, namely *control*, *memory hierarchy*, *host interfaces*, and *Design Space Exploration (DSE)*. First, Spatial provides multiple control structures to enable the designers to describe the accelerator architecture briefly, e.g., finite-state machines, streaming, parallel. Then, the memory hierarchy supplies various memory templates to abstract and yet control data allocation on both on-chip and off-chip memories. A designer can implement with Spatial code both accelerator and host, abstracting the underlying communication interfaces. Finally, even though the Spatial compiler automatically optimizes the control and memory constructs according to statically inferable information, Spatial also provides a DSE engine based on the HyperMapper [230] machine learning framework.

3.3.3 Intermediate Infrastructure for DSLs

The third and last cluster covers those solutions that propose an intermediate layer lying between the DSL and the RTL/HLS code. Developers can rely on such a layer to design new DSLs or extend existing ones to support FPGAs as target devices. In this way, the intermediate infrastructure further decouples the code development from the hardware-related translation and optimization, increasing the compilation flow modularity.

Calyx: Aiming at combining abstraction and control flow details of imperative HLS and HDL structural details and high performance, Calyx is an intermediate language that provides a shared compilation infrastructure to quickly design and deploy accelerators in Verilog [231]. Calyx provides a higher level of abstraction than IR for RTL languages [232] and grants

precise control over scheduling logic generation, borrowing the decoupling of algorithm and scheduling from Halide [131] while explicitly representing low-level resources. Calyx is not tied to any specific hardware design methodology, and it provides a general infrastructure to let new DSL-to-RTL be fastly prototyped. However, Calyx does not provide any target-specific optimizations, e.g., mux cost in ASIC or FPGA designs [231], and does not guarantee any feasibility on the design implementation.

3.3.4 Summary

Table 3.3 summarizes the main features of the tools. As in the second column, most owners are universities, which usually developed such DSLs for internal research projects or collaborations with other institutes/companies. However, some of these languages are currently not maintained anymore, according to the last updates on their repositories. In terms of the input language, while some developers designed a new language from scratch (e.g., Rigel, RIPL), the majority leverage existing languages and adapt their structure to target FPGAs, especially when considering the first two clusters. On the other hand, developers can exploit the intermediate infrastructure of the third cluster to either build a new language that directly supports FPGAs or add FPGA support to multiple existing languages. Moving to the domain, the first cluster of DSLs mainly focuses on image and packet processing domains, in which FPGAs are particularly effective. The same holds for the second cluster, shifting the specialization from the application level to the architectural one. Finally, the third cluster broads the supported domains thanks to their agnostic approach. After processing the input code, most DSLs generate either RTL or code for HLS/ACS tools, whereas just one DSL offers a full flow that produces the bitstream as output. Therefore, they all need to interact with commercial tools. Consequently, the DSL infrastructure may require continuous updates to keep pace with FPGA tool changes. In this scenario, a modular approach similar to the third DSL cluster eases maintaining the language.

Summing up, the domain specialization introduces an additional abstraction layer that impacts the design for FPGAs at multiple levels. On the one hand, it reduces the steepness of the FPGA learning curve, mainly requiring domain knowledge to write the algorithm. On the other hand, it automates various analyses and optimizations that, otherwise, would take too much time in a multi-domain scenario, relieving the designer from this burden. Nonetheless, making a DSL successful is no easy task as, at first glance, many DSLs look similar, especially when considering the same do-

Table 3.3: *Comparison table of the presented DSLs for FPGA design.*

	Tool	Owner	Input Language	Domain	Output	Target Vendor	Other Relevant Features
Application Domain	P4-to-VHDL [155, 226]	CESNET a.l.e. Netcope Tech.	P4	Packet Processing	VHDL	Xilinx	-
	Hipacc [233, 234]†	Nürnberg Saarland	C++ DSL	Image Processing	OpenCL/C++	Intel, Xilinx	CPU, GPU
	Darkroom [60]†	Stanford	Terra	Image Processing	Verilog	Xilinx	CPU and ASIC support, automatic scheduling
	Rigel [235]†‡⋆	Stanford	Rigel	Image Processing	Verilog	Xilinx	Simulation by Terra
	RIPL [236, 237]†‡	Heriot-Watt University	RIPL	Image Processing	Verilog	Xilinx	Algorithmic skeletons
	P4FPGA [156]†	P4FPGA Project	P4	Packet Processing	BSV	Xilinx	-
	Halide-HLS [126]†	Stanford, Berkeley	Halide	Image Processing	C/C++	Xilinx	Automatic integration (APIs, Drivers)
	P4HLS [238]†	Montréal	P4	Packet Processing	C++	Xilinx	-
	HeteroHalide [239]†‡	UCLA	Halide	Image Processing	HeteroCL	Intel, Xilinx	Multiple architectural backends
Architectural Domain	SuSy [240]	Cornell, Intel, UCLA	URE	Systolic Arrays	OpenCL	Intel	Exploits Halide OpenCL generation framework
	Spatial [228]†‡⋆	Stanford	Spatial	Application Accelerators	Chisel	Intel, Xilinx	DSE, CGRA and ASIC support
	SPGen [241]	Riken Center, University Tsukuba	OpenACC	Streaming Processing	OpenCL,SPGen	Intel	Still Experimental
Intermediate Infrastructure for DSLs	Infrastructure for Delite-based DSLs [242]	EPFL, Stanford	OptiML	Delite Languages	Bitstream	Xilinx	Dynamic Memory Allocation
	DHDL [101, 243]	Stanford	DHDL, other DSLs	Parallel Patterns	MaxJ	Intel, Maxeler's DFE	Cycle count and area estimates, DSE
	FROST [125, 244]	PoliMI, MIT	Halide, Tiramisu	Data-Parallel	C/C++	Xilinx	Scheduling commands
	LIFT [245]†‡	Edingburgh Glasgow University	LIFT	Functional patterns	VHDL	Xilinx	Host generation, Automatic DSE
	AnyHLS [246]†‡	Erlangen-Nürnberg	Impala	Any	C/C++/OpenCL	Intel, Xilinx	Partial evaluation, shallow embedding
	Calyx [231]†‡⋆	Cornell University	Systolic Arrays, Dahila, TVM	Any	Verilog	Xilinx	Can support any DSL

† Open-source ‡ Last update in the last years (2020-2021) ⋆ Maintained, to the best of authors' knowledge

main. We believe that maintaining a language and building a community around it, especially if open source, is paramount for its success. Likewise, being a vendor-agnostic DSL and supporting heterogeneous architectures help cover a broader range of designers. Even though this aspect potentially implies a higher complexity, a modular structure may ease addressing it thanks to the decoupling of the internal components. Finally, despite the automatic optimizations, we believe that a DSL would be more comprehensive if it exposed a manual optimization flow (e.g., a scheduling language like in Halide) that expert designers may exploit to hand-tune their designs.

3.4 Final Remarks

FPGAs are becoming increasingly pervasive in the computing landscape, from small low-power embedded systems to large-scale datacenters [25, 64, 247]. Similar to what happened with programming languages and frameworks for CPUs [248], tools for FPGA hardware design evolved according to the developer's needs and target contexts. Indeed, modern HDLs and HLS tools offer a new level of abstraction and productivity than (System)Verilog and VHDL. On the one hand, hoisting of abstraction level allows to reduce the design time, and facilitate IP reuse, customization, and verification. On the other hand, it makes the FPGA learning curve smoother for non-hardware designers. The possibility to use high-level languages like C, C++, and OpenCL is for sure advantageous over HDLs; however, such languages are not natively designed to describe hardware. Therefore, DSLs emerge to overcome these drawbacks and exploit the narrowing of the domain specialization. In this way, hardware developers benefit from specialized toolchains and infrastructures, optimized architectural templates, and expressing computations more simply and intuitively.

Overall, the community efforts push towards a constantly increasing abstraction of FPGAs' programmability to ease their usage and open to a broader public. Indeed, many of the described toolchains and other domain-specific toolchains that do not leverage a DSL as this Chapter intend (such as machine learning ones [41, 152] or general IR [249]) push the FPGA democratization. Although these toolchains cover several application fields — acceleration mainly —, we believe a low-level component (HDL) is still necessary to devise a complete design experience. In conclusion all these three abstraction efforts are necessary as they cover different aspects of digital design; thus, we believe the community will push the research in the FPGA programmability field, from low-level RTL design to domain-specific abstractions

CHAPTER 4

A Framework for Customizable FPGA-based Image Registration Accelerators

This Chapter describes a design automation framework for highly specialized and optimized stream-dataflow accelerators for Image Registration (IRG). The following Chapters exploit this design automation methodology and overall abstraction framework at the foundation of their design and deployment process.
IRG is a highly compute-intensive optimization procedure that determines the geometric transformation to align two images given their time instance, angle acquisitions, and sensor types differences. Though hardware accelerators are a promising solution for this domain, most implementations are either closed-source or tailored to a specific context, limiting their application to different fields. For these reasons, this Chapter proposes an open-source hardware-software framework to generate a configurable architecture for the most compute-intensive part of registration algorithms, namely the similarity metric computation Mutual Information (MI). It comprehends a complete stack, from the hardware layer to the software one, that enables easy application interfacing to an iterative design process.

Image Registration (IRG) is the process of identifying the parameters of the geometrical transformation matrix that allows the correct overlap of two or more images acquired in different conditions or time instants [250]. It always relies on a three-block structure: the transformation model, the optimization method, and the similarity metric [251]. The transformation model is clustered in rigid, which allows rotranslations, scaling and shearing, and non-rigid, which allows object deformation [252]. The optimizer searches the transformation space to find the optimal parameters of the geometric transformation with different strategies [253, 254]. Finally, the most used similarity metric is the Mutual Information (MI) [255–257] due to its robustness and reliability [258]. The achievement of a correct registration is strictly correlated to multiple iterations of the three blocks, where the similarity metric has proven to be the most compute-intensive, working directly with all the data contained in the employed images [259]. Unfortunately, even though improvements have been done, the majority of the hardware-available solutions are closed-source and, generally, tailored to a specific scenario, highly reducing, if not completely preventing, the users from customizing them.

Within this context, this Chapter proposes an open-source hardware-software framework for multi-modal IRG [260]. The proposed framework automates the design and synthesis of a customizable FPGA accelerator that targets the compute-intensive MI calculus. Thanks to the various customization parameters the framework exposes, the user can quickly explore the design space, tune the features of the MI accelerator, and tailor it to multiple case studies with different requirements. On the other hand, our solution offers high-level Application Programming Interfaces (APIs) based on the PYNQ framework [61] to easily integrate the accelerator within Python applications. We evaluated various versions of our accelerator on multiple Field Programmable Gate Arrays (FPGAs) achieving a speedup up to $2.86\times$ against an optimized MATLAB implementation, and remarkable performane and energy efficiency results against literature approaches.

4.1 Background

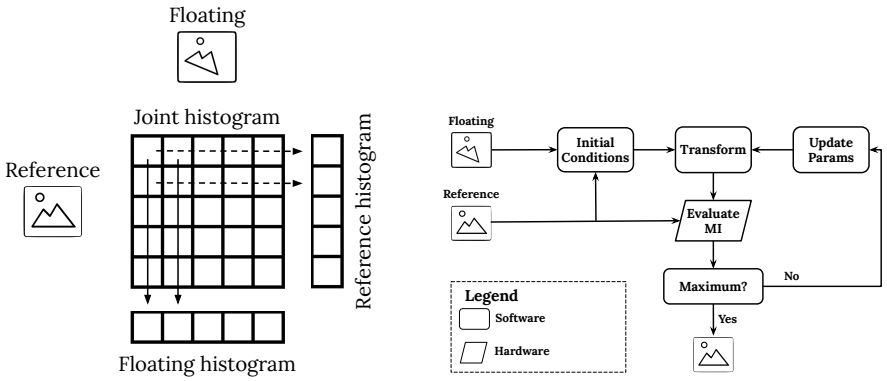
IRG is a highly employed procedure in various fields [45, 261–263] and, therefore, different implementations have been proposed during the years. This procedure processes data ranging from satellite images of the Earth to medical images both anatomical and functional [259]. Based on the image types, we can distinguish between mono- and multi-modal registration; the former works with images taken from the same device, while the latter em-

ploys images taken from different sensors [250]. An additional distinction can be done between feature-based and intensity-based approaches [255]. The first one requires the identification of relevant features from the images, giving a higher visual certainty of the reached correspondence, but it is not applicable when the landmark points identification is not trivial [250], as in multi-modal applications. Intensity-based algorithms exploit heuristic solutions that compare the intensity distribution of the images and decide whether or not the images are correctly aligned according to a similarity metric [255]. In this work, we consider a multi-modal intensity-based registration solution that can exploit either the (1+1) Evolutionary or Powell's optimization methods with the MI similarity metric to find the optimal parameters of the affine transform between medical images.

A wide range of algorithms employs **MI** quantity. In imaging, it is an essential similarity metric for IRG [264], in the genomics field it is exploited in phylogenetic [265] and relevance networks [266], as well as in the training of Hidden Markov Models and features selection [267, 268].

MI is a concept borrowed from the Information Theory that relates to the concept of entropy, and it is a measure of the statistical dependence of two random variables X and Y [269]. In the proposed scenario, the two variables are represented by images, where we can identify a *reference* and a *floating* one. In particular, we want to align the floating image to the reference image. From a mathematical point of view, MI describes how similar the joint entropy $H(X,Y)$ is to the two single entropies $H(X)$ and $H(Y)$, as in $MI(X,Y) = H(X) + H(Y) - H(X,Y)$. Therefore, an essential step is the definition and computation of the different entropies, as defined by the Shannon's equation: $H(X) = -\sum_{x \in X} P(x) \log P(x); H(X,Y) = -\sum_{x,y \in X,Y} P(x,y) \log P(x,y)$, where $P(x)$ and $P(y)$ are the marginal probabilities, and $P(x,y)$ is the joint probability. In the imaging field, the probabilities come from the histograms of the images, while the joint one from the joint histogram. Starting from the input images, we compute the joint histogram, which is a square matrix of $N \times N$, where N is the number of gray levels in the images [270]. The value of each element of the joint histogram $hist(x,y)$ is equal to the total number of voxels of X with intensity x corresponding to the voxels of Y with intensity y [269].

As shown in Fig. 4.1a, it is possible to exploit the joint histogram to efficiently obtain the single histograms of the input images. Indeed, by summing the rows and the columns of the joint histogram we obtain the reference and the floating histograms, respectively [269]. For the entropy calculation, we need to obtain the marginal and joint probabilities, which can be easily extracted by dividing each value of the single and joint his-



(a) *The relationship between joint and single histograms* **(b)** *High-level view of the workflow and its components*

tograms by the image dimensions in voxels. Referring to entropy equation, the last step is to apply the equations, hence, to multiply each probability by its logarithmic value, and, by accumulating them, we obtain the entropies. Finally, we combine all the entropy values to extract the MI, as in the equation.

4.2 Related Works

This Section contains an overview of the current literature with an in-depth focus on hardware-based solutions for multi-modal registration, being the scope of the proposed case study. In a pure software scenario we should mention SimpleITK [271,272], OpenCV, and the MATLAB Image Processing Toolbox [273]. While the first two are open-source, the last one is a licensed closed-source product. On the other hand, MATLAB is easy to use, while OpenCV provides fewer functionalities for IRG compared to the others, and SimpleITK, even though it can be easily used with Python, provides, like all the others, little control on small details. The literature contains several works exploring FPGA- and GPU-based solutions for multi-modal IRG to overcome the limits of pure CPU implementations [253]. From an algorithmic point of view, based on [259,274], it is possible to conclude that the most compute-intensive part is typically the calculus of the similarity metric. For this reason, in [259], the authors develop an FPGA-based accelerator to compute the similarity metric, namely the correlation, and the transformation model, an affine one, to register iris eye images through the Simplex optimizer. Besides, [274] proposes an FPGA-based approach, based on [275], to compute the MI value for multi-rigid registra-

tion, with a single computation of the MI for 7-bit 256×256 images taking around 0.26 seconds. The authors explored such an approach through an extensive design space exploration in [276]. On the other hand, [277] exploits GPUs to accelerate the sole joint histogram for brain images registration, based on MI, with a presorting strategy of the pixels. In [278], authors present a 3D MI-based IRG algorithm, using bitonic sort and count, which they tailor for GPU to achieve the best performance out of rigid transformation and Powell optimizer. Based on [278], [279] develops a CUDA-based optimization strategy for deformable registration fashion that aims at optimizing joint histogram – and then Normalized MI – and gradient computation, though exploiting some pre-computation mechanisms and dataset-specific techniques that reduce the computational requirement.

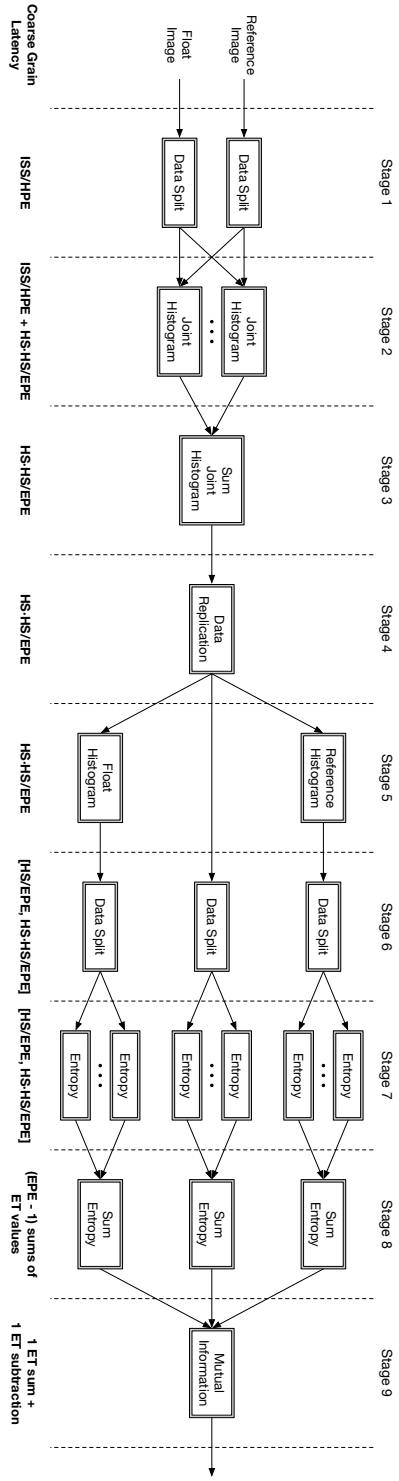
As discussed so far, hardware-based solutions are desirable in IRG, given the high-intensity workload. However, FPGA-based solutions are generally closed-source and not customizable by the final user, while GPUs are not customizable architectures at all, and are known to be power hungry devices. Based on these considerations, with this work, we propose an open-source hardware-software framework for IRG that exploits a customizable FPGA-based accelerator for the computation of MI, easily reusable in several IRG algorithms or even different fields of applications, such as phylogenetic [265] and features selection [268]. Moreover, being based on the PYNQ framework, our APIs are easily employable through Python, resulting transparent to the end-users. Finally, the overall hardware-software system is deployable on different FPGAs from embedded to high-end.

4.3 Proposed Design Methodology

The IRG procedure we consider in this work is an intensity-based multimodal algorithm, and its three main building blocks are affine transformation, Powell's and (1+1) Evolutionary optimization methods, and mutual information similarity metric. Since the similarity metric is the most compute-intensive part [259, 274] and multiple fields of application benefit from a MI accelerator (e.g., genomics computations), we present a framework to produce an architecture able to perform the MI computation. Fig. 4.1b shows the steps of the whole workflow to register two images and which part is offloaded to the hardware accelerator (Evaluate MI).

4.3.1 Framework Overview

This work proposes a framework to assist users in the generation of different versions of a hardware accelerator for MI calculation according to their



(a) *High level view of the architecture for MI computation and its latency (latency parameters defined in Tab. 4.1)*

needs. It is important to note that, even though this work focuses on IRG, the MI calculation is an essential part of many fields of application. Therefore, other contexts, where users may have different requirements, could benefit from MI acceleration. For instance, the final user might choose to sacrifice performance to save resources or to target a high-end scenario where performance is the main goal. For this reason, we devised our framework as an open-source solution capable of guaranteeing a high level of flexibility in the generation of the MI accelerator.

Our framework provides different *customization parameters* to assist the user in the exploration of the design space and to tune the multiple features of the MI accelerator. In particular, such parameters have a direct impact on the performance the resulting accelerator can achieve, as well as its resource consumption. Section 4.3.3 accurately describes the customization parameters. After selecting the parameters, the framework applies the requested customizations to the base structure of our FPGA-based accelerator (more details in Section 4.3.2). The result is a tailored design devised to perform MI calculation suitable for High-Level Synthesis (HLS) tools. Besides, the framework generates specific scripts to automate both the HLS process and the synthesis flow. In particular, given an either embedded or high-end target device, the framework determines all the required steps towards the bitstream generation. Finally, once the synthesis process is over, the user can leverage on the transparent Python APIs the framework supplies, and easily integrate the accelerator usage within applications based on the Xilinx PYNQ framework [61]. Indeed, we provide Python APIs able to handle all the accelerator configurations independently, e.g., caching or not caching (see Section 4.3.3), hiding the differences of the embedded or high-end device, and in some cases to measure the power directly on-board. A new IRG procedure, or an algorithm that employs MI computation, can exploit our accelerator with four simple additional steps: prepare the buffers with the data, start the MI computation, collect the results, and free the buffers.

4.3.2 Accelerator Architecture

Our accelerator adopts a dataflow computational model for en efficient multi-stage pipeline. Fig. 4.1a depicts the proposed pipeline and reports the coarse grain latency of each Stage, which we will analyze in Section 4.3.4. In particular, we identified two macro Stages within the pipeline. The first macro Stage (Stage 1 to 3) mainly consists of the joint histogram computation, while the second one (Stage 4 to 9) regards the entropy computations for MI calculation. Finally, each Stage is internally pipelined and streams

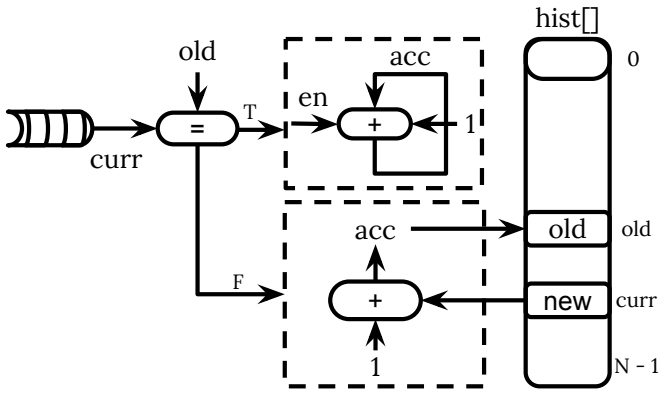


Figure 4.1: *A generic Processing Element optimized for histogram computation. In the case of a joint histogram, curr is the current index computed for a flattened 2D array.*

the data to the following one through FIFOs.

Input fetching: At the very beginning of the execution, the architecture fetches the two input images (reference and floating). This step depends on the device physical memory ports and the available bandwidth. Since not all the FPGA-based devices offer multiple memory ports, to be as generic as possible, we consider a case with a single memory port that is multiplexed (in case of multiple ports available, accelerator can be replicated according to the number of ports). However, this scenario may harm the accelerator performance, especially in the case of memory-bound designs, as the current one. Hence, it is paramount to properly design the accelerator according to the bitwidth of the memory ports and the memory bandwidth. A solution to alleviate such a problem could be to prefetch one or both images on the local memories. This is particularly suitable for algorithms, like IRG. Indeed, to register two images, the reference image does not change throughout the whole optimization procedure, while the floating one continuously changes. Therefore, if the target FPGA has enough on-chip memory and we apply this feature, our architecture first prefetches the reference image and then reads the floating one in a streaming fashion; otherwise, it reads both images simultaneously.

Stage 1-3: Assuming an input image bitwidth (IBW) of 8-bit, and a 32-bit memory port bitwidth (MBW) on the target device, we can pack more data (MBW/IBW) per single memory transfer, i.e., 4 pixels (8-bit wide each). Thus, this Stage takes the data coming from either the off-chip

Table 4.1: *Customization Parameters of our Architecture*

Parameter	Description
CACHE	support caching of one input image
IBW	input bitwidth of the single data, e.g., pixel
MBW	input port bitwidth, or memory port bitwidth
ISS	maximum input stream size value
HS	single histogram size
HPE	number of parallel histogram PEs, depending on MBW value
EPE	number of parallel entropy PEs
ET	data type precision of entropy computation
NCORE	number of parallel cores

memory or the on-chip one and splits them to multiple Processing Elements (PEs) performing the joint histogram. HPE defines the number of parallel PEs.

Stage 2 and Stage 3 compute the joint histogram of the two images with a map-reduce approach [280]. During Stage 2, the PEs receive two streams of unpacked data, from both the reference and floating images. As shown in Fig. 4.1, each PE creates its local histogram (in this case, a joint one) and stores the histogram values in BRAM. The input streams generate the joint indices of the joint histogram positions in which an increment by one occurs. As stated in Section 4.1, the joint histogram counts how many times a couple of intensities i, j appears, with i, j belonging to the reference, and the floating images, respectively. Practically, the images flow through Stage 2, which increments the intensity at position i, j. To avoid RAW hazard in the computation, each PE is optimized to accumulate the current intensity on a register while i, j are the same, on the contrary, if the current i, j are different from the previous ones, the PE reads the new value to accumulate from BRAM and writes back the previous one. As soon as the joint histogram is ready, this Stage subsequently sends it out to the following one. The output stream contains multiple histogram values packed together. EPE indicates the number of packed values. Stage 3 reduces the parallel computed joint histograms by summing the values within the HPE input streams. The adopted map-reduce approach improves parallelism of the joint histogram computation, therefore the latency, at the cost of storing HPE different joint histograms.

Stage 4-9: As stated in Section 4.1, once the joint histogram is computed, it is possible to derive the separate histograms of the input images and their relative probabilities. This is crucial for the entropy computations, as the Shannon's formula of the entropy revolves around the sum of the probability times the logarithm of the probabilities (see section 4.1).

For this reason, Stage 4 replicates the joint histogram stream three times, so that Stage 5 extracts the histograms of the reference and floating images by reducing per rows or columns, respectively.

Stage 6 unpacks each input stream to EPE ones. Stage 7 is in charge of computing the entropy, starting from the three histograms. This Stage computes Shannon's entropy for the single input (section 4.1), and it is the only one that requires `floating-point` values. To limit the number of floating-point operations, we defer the scaling of each input (to retrieve the probability) to Stage 9. Thus, we only need `floating-point` values for log operations. In particular, this Stage relies on either IEEE 32-bit `floating-point` or custom bitwidth `fixed-point` values as the required data type. Stage 8 accumulates the partial entropy and computes the final ones. Finally, Stage 9 receives the three entropy values, computes the MI value for the two input images, and writes it back to the host.

4.3.3 Assisted Exploration of Design Configuration Parameters

Our whole design is highly customizable according to the target scenario, different application requirements, and the target platform. $CACHE$ parameter indicates whether the architecture can cache one image (like the reference) into BRAMs (or URAMs when available). Considering the input memory port bitwidth (MBW) and the input data type bitwidth (IBW), the joint histogram part is parallelizable through $HPE = MBW/IBW$ PEs, which impacts on the architecture memory footprint. Indeed, each PE stores a joint histogram whose size is $HS \cdot HS$, where $HS = 2^{IBW}$ is the size of a single histogram. Besides, a user can deploy the architecture configuration that fetches a maximum input stream size (ISS) per iterations of different sizes, e.g., a 512×512 reference and 512×512 floating. Likewise, the datapath is customizable to different input data types, as 8-bit pixel or 16-bit pixel. It is worth noticing that scaling to larger images, e.g., $ISS = 2048 \times 2048$, influences resource usage and image transfer times from the main memory. In particular, ISS slightly impacts the bitwidth of joint histogram elements, while it significantly affects BRAM/URAM usage of caching designs. Different parameters affect the design of the second macro Stage. EPE describes the number of histogram values coming from Stage 2 packed together and, consequently, the number of parallel entropy modules per histogram. Another parameter is the data type (ET) used for the entropy computation. The data type can be either 32-bit IEEE `floating-point` or custom bitwidth `fixed-point`, which may introduce errors in the entropy due to the precision loss. In particular, the

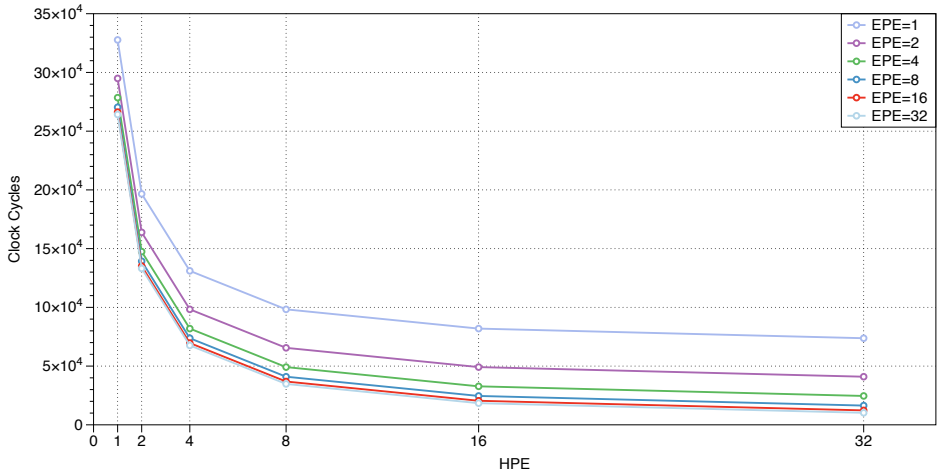


Figure 4.2: *Estimation of the proposed architecture latency according to the* HPE *and* EPE *parameters.*

bitwidth of `fixed-point` values depends on ISS. Finally, it is possible to deploy a multi-core version of the architecture to increase run-time performance. More specifically, we can instantiate $NCORE$ accelerators and connect each one to a different memory port (one per port to avoid contention). This permits us to perform multiple IRGs in parallel. Tab. 4.1 summarizes all the customization parameters, along with their description, as proposed in this Section.

4.3.4 Architecture Latency

The parameters in Tab. 4.1 enable to tune the proposed architecture according to the user's requirements. Starting from these parameters, we can analyze the latency of a given instance of our architecture and evaluate its theoretical performance. Fig. 4.1a reports the latency breakdown of each Stage of our design, while its formulae model latency at a steady-state for a coarse grain clock cycles estimation (without accounting pipeline warm up and off-chip memory bandwidth). However, the memory port bit-width (MBW) is a relevant parameter of our design, since it impacts on Stage 1 and 2 latencies and HPE parameter. In particular, given HPE and ISS, each block of Stage 1 takes ISS/HPE clock cycles to read the input from the off-chip memory and split it in HPE streams of data. Similarly, each block of Stage 2 receives a stream of data and computes a partial joint histogram in ISS/HPE clock cycles. Then, these blocks write the partial joint histogram to the output FIFO in $HS \cdot HS/EPE$ clock cycles. Since

these two computations within Stage 2 cannot overlap, the latency of each block of Stage 2 is the one reported in Fig. 4.1a. Stage 3 reads the incoming HPE streams, sums the partial joint histograms, and writes the complete joint histogram to the output FIFO in $HS \cdot HS/EPE$ clock cycles. The following Stage 4 takes the same amount of clock cycles to read the joint histogram and replicate it three times. The computations performed in the two blocks of Stage 5 differ, but they share the same coarse grain latency. Starting from Stage 6, the latency of each block varies, even though they are identical internally, as reported in Fig. 4.1a. Indeed, the blocks directly connected to Stage 5 read a smaller amount of data than the one directly connected to Stage 4. In particular, the blocks of Stage 6 directly connected to Stage 5 receive a single histogram, while the one directly connected to Stage 4 the joint one. Consequently, the same holds for the blocks in Stage 7. The output of each block of Stage 7 is a single ET value. Thus, the latency of the blocks of Stage 8 is almost negligible, as they only sum the incoming values. Likewise, Stage 9 reads the three input entropies and outputs the mutual information in few clock cycles.

The proposed architecture works in a dataflow fashion, and each stage is internally pipelined. Hence, the coarse grain latency of a given instance of our architecture is $ISS/HPE + HS \cdot HS/EPE$. This value mainly depends on both the joint histogram calculation and its entropy computation. Specifically, considering a specific ISS and HS, both HPE and EPE provide a theoretical performance boost that scales as $1/x$. Fig. 4.2 shows how the coarse grain latency scales with $ISS = 512 \times 512$ and $HS = 256$. We extracted these values via the cycle-accuracy cosimulation of Vivado HLS. Fig. 4.2 does not take into account the memory bandwidth, for it does not illustrate the effects of caching. The values of HPE and EPE are a power of 2 for the sake of simplicity. In this case, HPE has a greater impact on the estimated latency than EPE, as $ISS > HS \cdot HS$.

4.4 Experimental Setup and Results

The proposed framework generates a C++ accelerator architecture suitable for HLS tools for Xilinx Vivado HLS 2019.2 toolchain. We target four boards from different scenarios: A Pynq-Z2 board based on the Xilinx Zynq SoC, an Ultra96 v2 board powered by a Xilinx MPSoC Ultrascale+ ZUEG3, a Zynq UltraScale+ MPSoC ZCU104, and an accelerator card, namely an Alveo u200 board with a Xilinx Ultrascale+ XCU200. These boards range from low-power embedded devices (Pynq-Z2) to high-end accelerator cards (Alveo u200). Through the scripts generated by the frame-

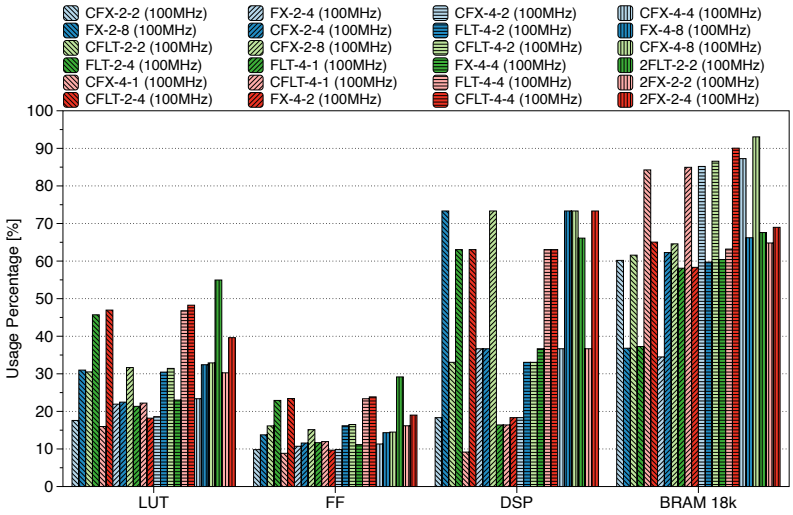


Figure 4.3: ***Ultra96*** *resource utilization. Resources: 70560 LUT, 141120 FF, 432 BRAM 18k, 360 DSP*

work, we employ the Xilinx Vitis and Vivado HLx toolchains to generate the bitstream, both version 2019.2. The IRG application is implemented in Python in a multi-threaded version, and the hardware-software interfacing part is handled through our Python APIs, which interact with the PYNQ [61] framework v2.5 for both the embedded and the high-end devices. The host processors are a dual-core ARM A9 (Pynq-Z2), a quad-core ARM A53 (Ultra96 and ZCU104), and a quad-core Intel i7-4770 (Alveo u200). While the first three reconfigurable fabrics are on the same chip with the host and share the DDR, the Alveo is connected through PCIe to the host device. We evaluated the proposed solution with a stack of 247 Computed Tomography (CT) images, with a dimension of 512×512 pixels, and a corresponding number of Positron Emission Tomography (PET) ones, resized from 128×128 pixels to a dimension of 512×512 pixels, from the medical field, each down-scaled to 8-bit wide[1] [281, 282]. Each image has various misalignments depending on the patients' movements or acquisition protocols. We compare our solutions against both works available in the literature and an optimized state-of-the-art MATLAB implementation [273], i.e., the one available in the MATLAB 2019b Image Processing Toolbox, which exploits as many cores as are available, and can be further optimized thanks to the Parallel Computing Toolbox, running on a dual-core Intel i5-7267U CPU and a 40 core Intel Xeon Gold 6148

[1] Patient: C3N-00704, Study: Dec 10, 2000 NM PET 18 FDG SKULL T, CT: WB STND, PET: WB 3D AC)

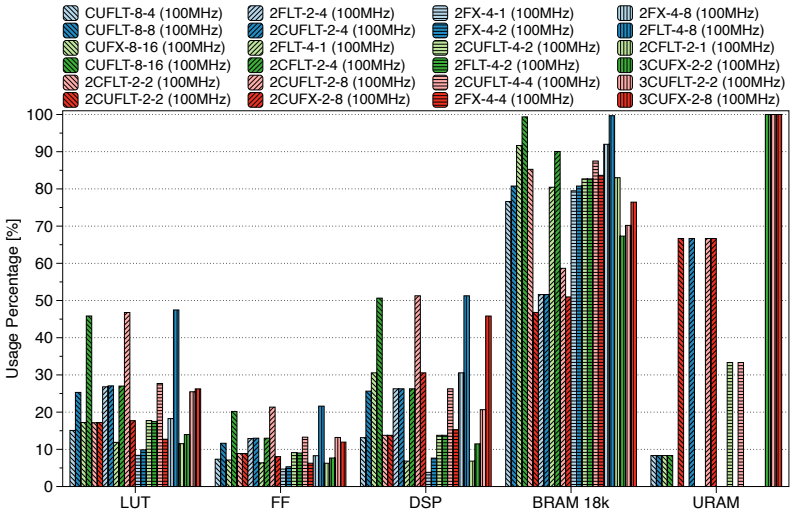


Figure 4.4: ***ZCU104*** *resource utilization. Resources: 230400 LUT, 460800 FF, 624 BRAM 18k, 1728 DSP, 96 URAM*

CPU. Our solution and the MATLAB implementation share the transformations, the similarity metric, and the input dataset. With respect to the optimizer, we exploit both Powell and (1+1) Evolutionary, while MATLAB the (1+1) [254]. Powell optimizes one parameter at a time in a given range using MI, while the (1+1) evolves genetically from a parent transformation vector to a child using MI and a normal random generator.

4.4.1 Experimental Evaluation

We first evaluate the target hardware-software framework through a design parameter exploration across the several customization parameters of the proposed architecture on the Ultra96, ZCU104, and Alveo board. Then, we compare the execution times of our hardware designs to MATLAB, with and without the Parallel Computing Toolbox. Finally, we analyze the best performance of the four boards against state-of-the-art works. Given the target case study, the framework allows us to quickly produce several designs adopting $IBW = 8$ and $ISS = 512$ for all the considered devices. We consider MBW as the product of IBW and HPE, thus it is not reported in any of the following analyses. We describe a configuration as the concatenation of the following parameters: $NCORE$ (if missing, we assume 1), $CACHE$ (if missing, no caching, otherwise C for BRAM caching, CU for URAM caching), ET (FLT for 32-bit `floating-point`, FX for `fixed-point`), HPE, EPE. For instance, FLT-2-1 describes a design

with 1 core, no caching, 32-bit `floating-point`, 2 HPE, and 1 EPE.

4.4.1.1 Resources Design Space Exploration

Fig. 4.3, 4.4, and 4.5 report the most relevant synthesis results for Ultra96, ZCU104, and Alveo we achieved throughout several runs. While we can see that the amount of LUTs and FFs used generally remains reasonably low, BRAMs and DSPs usage varies a lot based on the configuration. From the several single core versions, we can appreciate how BRAMs usage mainly comes from HPE scaling or the reference image caching, making BRAMs the critical resources of the proposed accelerator. Indeed, the devised PE for the joint histogram requires a local, though smaller, partial joint histogram memory. Therefore, increasing the level of parallelism dramatically boosts the performance, as we will discuss in the following Section, at the cost of higher impact on memory footprint. On the other hand, caching requires an on-chip memory able to fit $512 \times 512 \times 8$ bits. Besides, designs on Ultra96 using caching require a significant amount of BRAMs, while ZCU104 and Alveo board not only have more BRAMs but also URAMs in the reconfigurable fabric. Thus, all the configurations exploiting URAMs (the ones with the U), drastically reduce the BRAM usage and pave the way to configurations that otherwise would not fit the FPGA. On the other hand, EPE mainly impacts on the amount of required DSPs (particularly when using 32-bit `floating-point` values) due to the usage of logarithms (we rely on the implementation available within Vivado HLS) and `floating-point` multiplications. In the case of `fixed-point`, the integer and decimal parts are a function of ISS, and here are 23 and 19 bits. We analyzed the impact of `fixed-point` precision on the MI calculation and measured an MSE of 3.46E-10 compared to `floating-point` (100 tests with random inputs). Considering the multi-core version, we should notice how the Ultra96 scales to few cores, while ZCU104 and Alveo fabrics can scale to multiple cores with different combinations of HPE and EPE. Upscaling the core design number is limited by BRAM, DSP, and the number and wideness of physical DRAM ports. For this reason, increasing the number of cores on Ultra96 or ZCU104, which have a single DDR with a 32-bit port, creates contention and leads to performance degradation, as there are no more logical resources. On the other hand, the Alveo u200 has 4 DDR memory banks, each of which has a capacity of 16GB and 64-bit ports. Thus, on the Alveo, we exploit one core per memory bank whenever it is possible, and, as such, the HPE value multiplied by 8 gives the resulting bit-width the

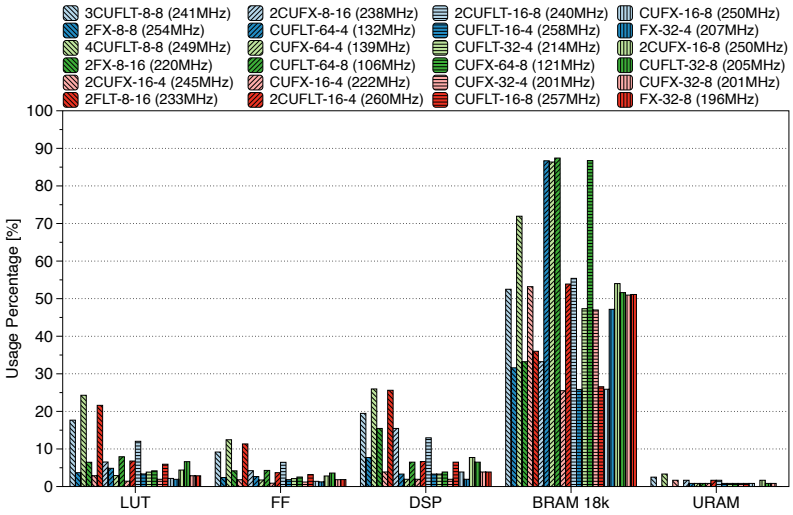


Figure 4.5: ***Alveo u200*** *resource utilization. Resources: 1019968 LUT, 2128354 FF, 3532 BRAM 18k, 960 URAM, 6833 DSP*

cores would require. It is worth noticing that, in this case, the data transfer via PCIe between the host and the Alveo board may become a bottleneck when using multi-core designs. Finally, while, for the Alveo designs, Vitis automatically scales all the design at the maximum frequency it can handle, and thus does not require manual intervention, Vivado does not. Hence, we synthesize Pynq-Z2, Ultra96, and ZCU104 designs at 100MHz, but then we exploit the PLL of the Processing System to hand-tune at run time the frequency and check the consistency of our results. As a result, Alveo designs run at the frequency reported in Fig. 4.5, while all the others at 200MHz.

4.4.1.2 Performance Analysis

We evaluate the architecture performance for an IRG application on the execution time of the single value of MI, the single IRG, and the overall stack of images. Fig. 4.6 shows how the single MI computations vary according to ET, HPE, EPE, and $CACHE$ while keeping a single core (multi-core would not impact the single MI computation), on the Ultra96 running at 200MHz. We can notice how, by increasing either HPE or EPE or enabling $CACHE$, we reduce the average execution time for a single computation. In particular, we can notice how HPE and EPE parameters impact the performance scaling in line with the latency analysis of Section 4.3.4. Conversely, when $CACHE$ is available, we have to account for both the caching time and MI computation time. Fig. 4.6 reports both the single MI

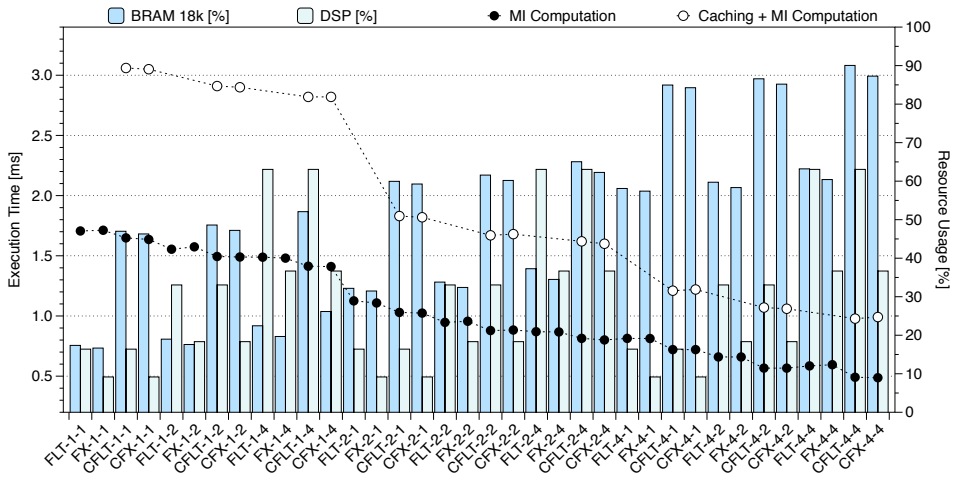


Figure 4.6: *Average execution time and resource usage scaling according to the main parameters that affect the architecture, i.e., latency HPE, EPE, ET, and CACHE. Standard deviation not reported as negligible (from* $2.61E-03$ *to* $3.12E-02$*).*

execution time and the aggregate one. Although this solution may be inefficient for a single MI computation, it helps to decrease the overall time when one of the two images does not change for several iterations, e.g., during the IRG process. On the other hand, Fig. 4.6 shows the direct connection between HPE and BRAM usage and between EPE and DSP usage. It is interesting to highlight that it is possible to distribute the level of parallelism between HPE and EPE and partially reduce performance while balancing resource usage (configurations FLT-4-1 and FLT-2-2). Finally, Fig. 4.6 shows that the FX data type significantly reduces DSP usage, as well as slightly improving both the average computation time and BRAM usage , hence enabling $EPE = 8$ or $EPE = 16$ configurations.

We also measured the average execution time per image, and the overall registration time for the entire stack of images. Fig. 4.7, 4.8, and 4.9 show how these times scale for the most significant configurations on Ultra96, ZCU104, and Alveo, respectively. We sorted the configurations in decreasing order according to the overall registration times of the (1+1) Evolutionary. In general, the most impacting factor in the execution time comes from the HPE value, which impacts the bandwidth and the time required to process and compute the joint histogram (one of the biggest bottlenecks). Another relevant factor is caching. Indeed, most of the configurations exploit caching to reduce the execution time required by the application. The entropy data type ET affects the execution time of the

Table 4.2: *Comparison with Related Works with Perf. measured as [*ms/MVoxels/iterations*], as proposed by [253], and Energy Eff. as [*(MVoxels · iterations)/(ms · kWatt)*]*

Arch.	Work	Transform	Metric	Optimizer	Hardware	Perf. (lower is better)	Power Eff. (higher is better)
FPGA	**FX-4-4**	Affine	MI †	Powell	PYNQ-Z2 (28nm)	49.90	8.02
	2FLT-2-2	Affine	MI †	Powell	Ultra96 (16nm)	11.02	12.52
	2CFLT-2-1	Affine	MI †	Powell	ZCU104 (16nm)	9.34	8.56
	FX-32-8	Affine	MI†	Powell	Alveo u200 (16nm)	1.78	18.52
	FX-4-4	Affine	MI †	1+1	PYNQ-Z2 (28nm)	0.42	979.76
	2FX-2-4	Affine	MI †	1+1	Ultra96 (16nm)	0.09	1534.00
	3CUFX-2-8	Affine	MI †	1+1	ZCU104 (16nm)	0.08	981.60
	FX-32-8	Affine	MI†	1+1	Alveo u200 (16nm)	**0.02**	**2058.98**
	[275]	Rigid	MI†	N/A	2xAltera 1K100 (n.a.)	101⋆	N/A
	[274]	MultiRigid	MI†	Simplex	Altera EP2S180 (90nm)	13.4⋆	N/A
	[259]	Affine†	Corr.†	Simplex	Zybo (28nm)	9.15⊙	N/A
GPU	[277]	N/A	MI†	N/A	FX 5800 (55nm)	39.04/1.07▽•	0.13/4.94‡
	[278]	Rigid	MI†	Powell	GTX 280 (65nm)	4.06⋆	1.04‡
	[279]	Nonrigid	NMI†	Grad. Desc.†	GTX 580 (40nm)	0.13▽◇	31.52‡
CPU	MATLAB	Affine	MI	1+1	Intel i5-7267U (14nm)	0.24⊗	176.97‡
	MATLAB	Affine	MI	1+1	Intel Xeon Gold 6148 (14nm)	0.14⊗	48.37‡
	MATLAB PAR. TOOL	Affine	MI	1+1	Intel Xeon Gold 6148 (14nm)	0.05⊗	145.93‡
	Simple ITK	Rigid	MI	Grad. Desc.	Intel Xeon Gold 6148 (14nm)	0.25⊗	27.01‡
	Simple ITK	Rigid	MI	Powell	Intel Xeon Gold 6148 (14nm)	2.51⊗	2.65‡
	Simple ITK	Rigid	MI	1+1	Intel Xeon Gold 6148 (14nm)	0.89⊗	7.46‡

†Implemented in hardware ⋆Computed from [253] ‡Computed with Termal Design Power (TDP) as power ⊙ Assuming maximum iteration of 500

▽ Exploits the binning to reduce joint histogram sizes • The first number includes presorting time ⊗ With maximum 100 iterations
◇ This value is the result of several dataset-specific approximations and preprocessing that reduce the computation to 1/6 and lead to misregistrations [279]

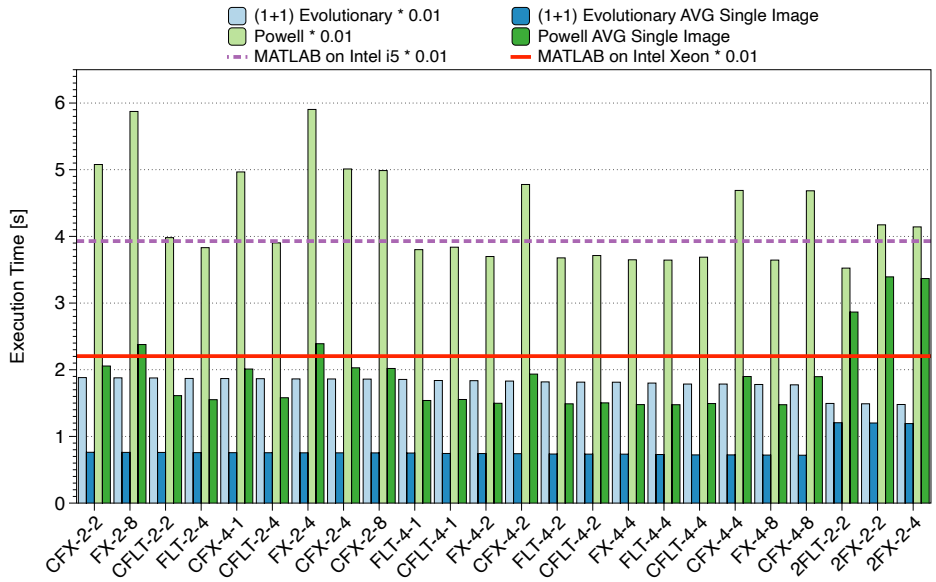


Figure 4.7: ***Ultra96*** *against* MATLAB *on both an Intel i5 and an Intel Xeon.*

registration process as well. Finally, we can notice how the $NCORE$ parameter has a limited effect on the execution times, due to both the unique physical port (on embedded boards) and the run-time overhead of Python when managing multithread applications. Thus, a careful balance of these parameters enables to optimize the application run-time. Considering (1+1) Evolutionary, Ultra96 and ZCU104 easily outperform the MATLAB reference on the Intel i5, reaching speedups up to $2.66\times$ and $3.21\times$, respectively. Similarly, both boards result faster than the MATLAB on the Intel Xeon without the Parallel Computing Toolbox enabled, and the best speedups are $1.49\times$ and $1.80\times$, respectively. On the other hand, the selected designs running on the Alveo u200 outtake the Intel Xeon by a factor up to $2.86\times$ (Parallel Computing Toolbox enabled) and $8.62\times$ (Toolbox not enabled), and $15.36\times$ against the Intel i5 (here not reported in the chart to ease the visualization). Moving to Powell's optimization method, we notice that this procedure is more sensitive to the entropy data type. Indeed, the configurations employing fixed-point data type take more iterations to converge Nonetheless, various designs of the Ultra96 and ZCU104 surpass MATLAB on the Intel i5. The same holds for multiple Alveo configurations against MATLAB on the Intel Xeon. However, it is crucial to note that we are considering two different optimization methods; indeed, Powell

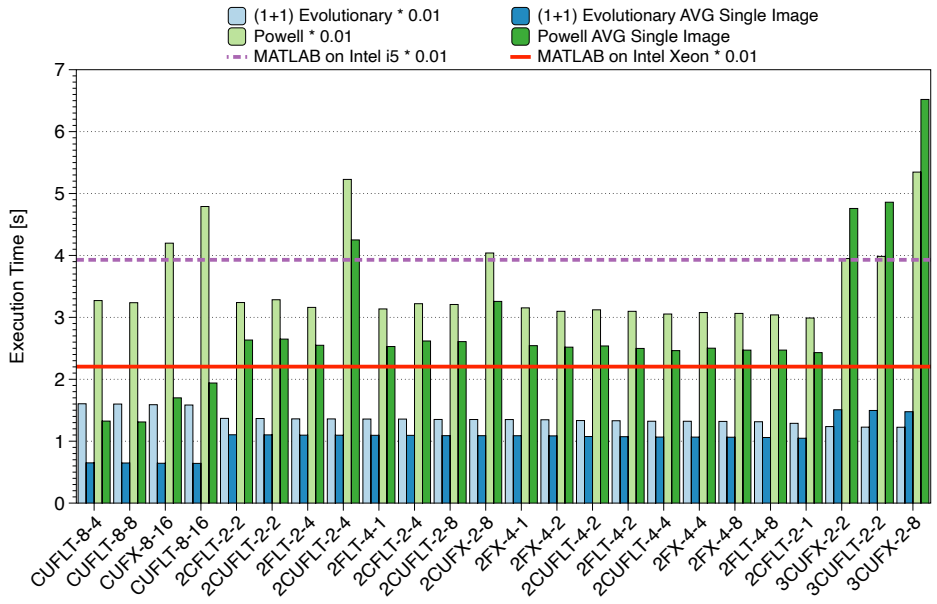


Figure 4.8: ***ZCU104*** *execution times of the whole application using both Powell and (1+1) Evolutionary, and their average time per image, compared against* MATLAB *on both an Intel i5 and an Intel Xeon. We scaled the execution times of* MATLAB *and our IRG applications by a 0.01 factor.*

optimizes a parameter at a time within a given range per iteration, while (1+1) generates a new child vector, comprehensive of all parameters, per iteration, hence the evolutionary algorithm requires fewer computations.

4.4.1.3 Accuracy Analysis

We compare the accuracy of our solutions against the MATLAB procedure with the Dice score metric, which evaluates how good is the overlap between two region of interests. Figure 4.10 shows a visual comparison example of the considered registrations and the gold standard We extracted the gold standard with a supervised semi-automatic procedure, based on the interactive MATLAB Registration Application exploiting the multi-modal registration model. To evaluate the Dice score, and hence the accuracy, we binarize both the gold standard and output images. As a consequence, if the border of registered structures are correctly overlapped to the gold standard, also the internal structures will be correctly registered, as the employed geometric transformation does not insert deformations. Based on this analysis, our top-performing Alveo implementation reached a mean

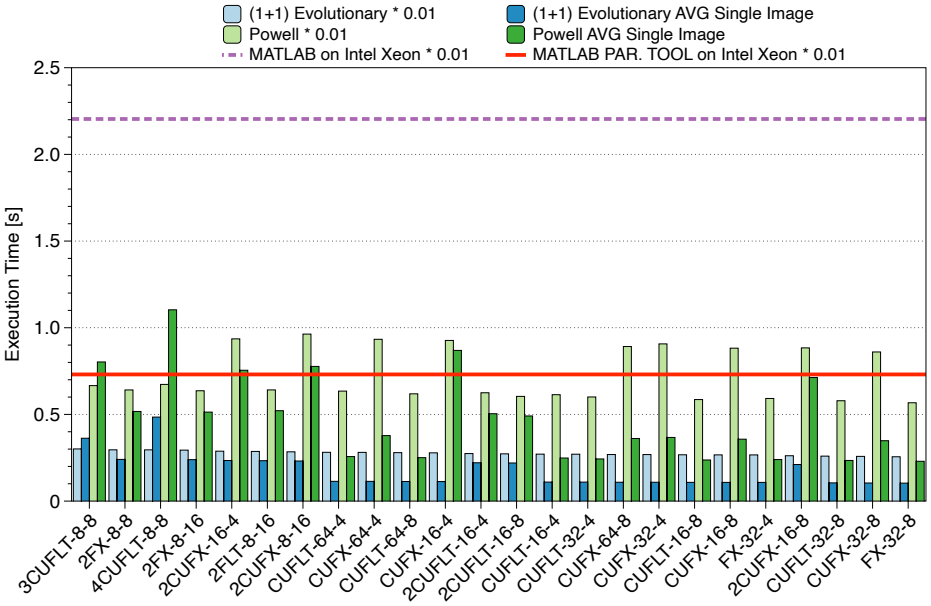


Figure 4.9: ***Alveo u200*** *execution times of the whole application using both Powell and (1+1) Evolutionary, and their average time per image, compared against* MATLAB *on an Intel Xeon. We scaled the execution times of* MATLAB *and our IRG applications by a 0.01 factor.*

Dice score of 94% and 78% with Powell and (1+1) respectively in line with both the optimized MATLAB implementation of 85% and the gold standard.

4.4.1.4 State-of-the-Art Comparison

In the IRG field, comparing against state of the art is extremely hard given the absence of a standard dataset, the availability of the source codes, the broad combinations of the IRG methodologies, and the different hardware platforms. For these reasons, we open-source our solution at this link [260]. FPGA-based approaches in the literature [259, 274, 275] all compare against their single-thread software implementation only. Besides, they mainly perform mono-modal IRG on 256×256 images, using Simplex as the optimizer. While [274, 275] use MI as similarity metric and deformable transformations, [259] uses correlation and affine ones. Conversely, we use a 512x512 multi-modal IRG with Powell as the optimizer, MI as the similarity metric, affine transformations, and we compare against a multi-threaded MATLAB optimized software. [253] shows the struggle to provide a standard performance metric to compare different works, and

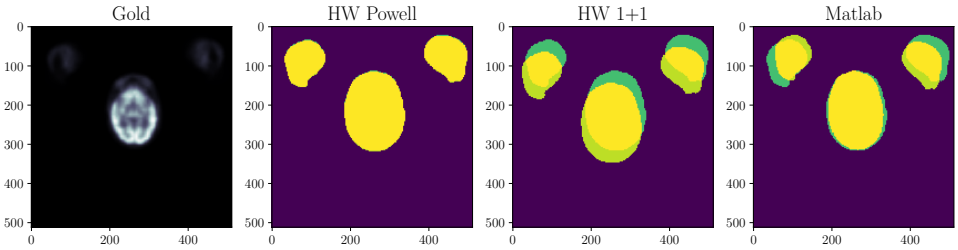


Figure 4.10: *A visual example of IRG results. From left to right, the gold standard, the overlap between gold standard (represented in green) and the registered image wih Powell, (1+1), and* MATLAB *(all in yellow)*

proposes $ms/MVoxels/iterations$ as a solution (the lower, the better)., where these parameters refer to the overall registration process of N images. Tab. 4.2 reports the metric proposed from [253], where we took numbers for [274, 275, 278], and we compute the same metric for this work and other works reported in Section 4.2. For many of those works, we have computed the performance according to the information available in [253] and to be as much fair as possible. We exploit the performance metric to compute the energy efficiency, last column of Tab. 4.2, as $1/(Perf. \times PowerConsumption)$. Considering Alveo and GPUs solutions, we account for the board power only, while the others also account for the CPU power, especially our embedded boards.

Regarding the FPGA-based solution, this work achieves better performance against all the selected FPGA approaches, reporting 0.02 and 1.78 $ms/MVoxels/iterations$, with FX-32-8 as configuration for the Alveo u200, respectively for (1+1) and Powell. By looking at the GPU-based approaches reported in Tab. 4.2, most of them rely on both precomputation techniques and binning strategies to reduce the computational load of the algorithm. While the former is mainly algorithmic and dataset dependent, exploiting binning levels makes the joint histogram computation, and the following ones, less time consuming, by sacrificing the accuracy. In this context, [279] reports encouraging numbers, though it seems they apply many precomputation and dataset-specific techniques without which the computation would be way more expensive. The authors also report various cases where the registration process fails to align the images. Indeed, our best performance result with the (1+1) on all the boards but PYNQ-Z2 are better than [279], without computation reduction techniques. Moreover, we outtake all the considered GPU implementation in terms of energy efficiency, with maximum of $65\times$ (Alveo (1+1) versus [279]). Finally, we have compared our implementations against MATLAB Image Processing

Toolbox and SimpleITK. As we can see from Tab. 4.2, MATLAB achieves better results compared to SimpleITK, and in particular it reaches results in line with our (1+1) top implementations, when the Parallel Computing Toolbox is enabled on the Xeon Gold. However, considering the energy efficiency, all the MATLAB implementations prove to be less efficient than our solutions with similar performance.

4.5 Final Remarks

In this Chapter, we presented an open-source hardware-software framework to automate the design and synthesis of a configurable architecture for mutual information calculus oriented to the possible constraints for the end use case. We evaluated our framework in both the embedded and high-end scenarios through the given transparent APIs, showing how our designs scale according to the parameters and how it provides remarkable performance. Indeed, we compared our accelerators against an optimized MATLAB version on an Intel Xeon Gold reaching a speedup of up to $2.86\times$ for the registration of 247 images and a Dice score of 94% and 78% with Powell and (1+1) respectively. We compared our performance with state-of-the-art approaches employing different techniques, overwhelming FPGA-based solutions, with the metric proposed in [253], and achieving remarkable power efficiency results.

CHAPTER 5

A Depth-First-based Domain-Specific Architecture for Efficient Regular Expressions Matching

This Chapter presents the background knowledge on Regular Expressions (REs), the literature of this domain, and the first of the two Domain-Specific Architectures (DSAs) for RE matching. In particular, this architecture, called TiReX, exploits an execution model similar to a Deterministic Finite Automata (DFA) with a depth-first approach and a backtracking mechanism. Although flexible on the set of searched REs, software-based solutions cannot fulfill latency or throughput requirements to analyze massive data volumes at a given power budget. Hence, many fields exploit hardware accelerators as an offloading engine for REs matching. However, various solutions rely on Field Programmable Gate Array (FPGA) reconfigurability to embed automata into the reconfigurable fabric exploiting so-called Stream-Dataflow Architectures (SDA) or In-Memory Architectures (IMA), leading to time-consuming updates of the REs to search. This Chapter exploits REs as sequences of basic instructions and the TiReX DSA for RE matching on FPGAs, employing the flexibility of a Software-Programmable Architectures (SPA) with specialized hardware mechanisms.

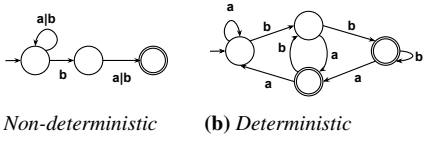
Many applications rely on determining whether a string obeys a specific text-based *pattern*, expressible in a compact and specialized way through Regular Expression (RE). RE matching determines whether a sequence of input characters belongs to the set of strings described by the pattern. It is an essential kernel [44] widely used in several fields: genome-protein analysis [283], text analytics [284], Intrusion Detection Systems (IDSs) [285], signature based detection [286–289]), natural language processing [290, 291], database queries [292,293]. This domain would benefit from a pattern matching engine with a good ratio of performance-per-watt and easy adaptability to address stringent time analysis requirements (e.g., IDSs [294]), fastly evolving fields (e.g., "Personalized Medicine" [295–297]).

Many researchers address these challenges by efficiently representing the automata that accept the RE-defined language [298, 299]. In contrast, others exploit specialized hardware (e.g., Application-Specific Integrated Circuits (ASICs), such as Ternary Content Addressable Memory (TCAM), Power Edge of Network™ (PEN), or spatial architectures, such as Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs) or Automata Processor (AP) [284, 300]) to build effective domain-specific solutions [301]. Given the computing and energy efficiency advantages over Central Processing Units (CPUs) [22, 25], and the higher adaptability against ASICs, several methodologies embed automata into the reconfigurable fabric of an FPGA. In this way, they leverage the reconfigurability to change the RE(s) to find [159, 160, 302]. Generally, the reconfiguration time of an FPGA is in the order of milliseconds, while generating a new bitstream requires from one to several hours, making real-time adaptation unfeasible [303, 304]. Thus, the embedding requires a database of ready-to-use bitstreams or bitstream regeneration if the pattern is new. On the one hand, sacrificing the run-time adaptability with automaton embedding achieves remarkable performance [305], on the other, flexibility carries a performance cost [306]. However, easily changing the searching pattern is an essential feature in many application fields [303, 304], where wasting a few microseconds leads to unsustainable performance degradation [307].

This Chapter proposes TiReX [48], a software-programmable Domain-Specific Reconfigurable Architecture (DSRA) for reconfigurable systems tailored to the REs domain. The DSRA overcomes the reconfigurability embedding issue exploiting the idea of using REs as a programming language for our custom Instruction Set Architecture (ISA) [48, 308–310]. This domain narrowing leads to an optimized microarchitecture for multi-character analysis easily adaptable at run-time to the REs to analyze (Section 5.3.3). The multi-core architecture can be software programmed to op-

Table 5.1: *Some meta-characters for Regular Expressions.*

Meta-Character	*Description*
xy	concatenation of x and y, same as writing $x\&y$
$x\|y$	alternation of x or y
$()$	priority encoding
x^*	repetition of zero or more x
x^+	repetition of one or more x
$x^{\{n,m\}}$	repetition of x from n times to m times
.	any alphanumeric character
$[x-y]$	character ranging from x to y



(a) *Non-deterministic* **(b)** *Deterministic*

Figure 5.1: *Different Automaton representation of the RE "*`(a|b)*b(a|b)`*".*

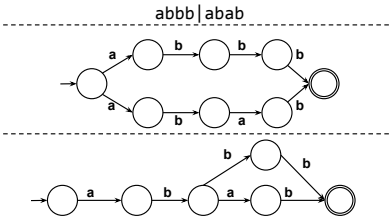


Figure 5.2: *From a RE "*`abbb|abab`*" (top) to its NFA (mid) and, finally, to DFA (bottom).*

erate in a multi-pattern single-stream of data or single-pattern multi-stream of data (Section 5.3.5). Moreover, our DSRA is easily deployable on different FPGA-based platforms depending on the target workload, e.g., embedded or high-performance (Section 5.5). The domain-specialization of TiReX pushes towards the energy efficiency required to address current computing challenges [301, 311].

5.1 Background on Regular Expression Matching and DFA-NFA Comparison

REs and Automata are a general execution engine for a wide variety of applications ranging from simple text searching to random forest execution [290].

REs are a declarative way of describing sets of character strings used to define Regular Languages [312], with meta-characters for specifying operations. Table 5.1 presents some of the most used RE operators in all the typical application scenarios (e.g., text-editors search functionality, software libraries for REs, web search engines). There are advanced operators on top of these symbols, which are nowadays involved in pattern match-

ing but not related to Regular Languages, such as backreference. In this work, we focus on classical REs. Finite State Automaton (FA) is a descriptive way of defining an abstract finite state machine that accepts particular string(s) defined by regular languages [312], hence also representable as (class of) RE(s). FA splits in Deterministic Finite Automata (DFA) or Non-deterministic Finite Automata (NFA) based on the possibility of being in one or more states simultaneously, i.e., deterministic vs. non-deterministic execution model. The DFA model requires only to keep track of the current state and moves into a new state every time it reads a character, resembling a depth-first exploration. However, some REs intrinsically carry a certain level of non-determinism. For example, the RE "`(abbb|abab)`" describes two alternative patterns that both start with the sub-string "`ab`". Therefore, the matching process does not know which pattern is matching until it evaluates the last two characters (either "`bb`" or "`ab`"). Even though it is always possible to move from NFA into a deterministic version via the power-set construction algorithm, this conversion can exponentially increase the number of states. Figures 5.1a and 5.1b show a simple example where the NFA representation (Figure 5.1a) requires less states. Though the NFA's execution model relies on a breadth-first-like approach and provides the best theoretical performance, it is equivalent to a DFA, which usually adopts a depth-first-like approach, on the class of acceptable languages [312]. Consequently, we need to adapt the matching algorithm to manage the states with alternative paths associated with the same character when using an NFA. A recursive implementation selects an alternative and, if wrong, it backtracks to the most recent "decision state" to evaluate a different path. This approach also requires reverting the portion of the string that was processed in the wrong path. In this way, the backtracking algorithm is simple but requires processing the input string multiple times. In the worst case, if the string does not match the RE, the algorithm must try all possible execution paths, leading to an exponential execution time [49]. An alternative approach with linear execution time has been proposed by Thompson [313] and used by Google in `RE2` [314], a RE software library, which is in use in many Google products like BigQuery[1] and Google Suite[2].

Software libraries, like the ones available in programming languages such as Python2.4 or Perl5.8, divide the RE into sub-expressions until the matching problem becomes manageable and then use a backtracking algorithm to evaluate the alternative paths [49]. **Example**: When applied to our example (top part of Figure 5.2), they first divide the pattern into two sub-expressions, namely "`abbb`" and "`abab`". This decomposition can be

[1] `https://tinyurl.com/cloudgooglecicero` [2] `https://tinyurl.com/googleregexsuite`

easily represented as the non-deterministic finite-state automaton shown in the mid part of Figure 5.2. When a character (e.g., 'a') is compatible with two or more sub-expressions (e.g., "abbb" and "abab"), the machine considers one sub-expression (e.g., "abbb"), keeping track of the possible alternatives. If the machine does not match the string (i.e., the RE does not *accept* the string), it needs to backtrack to the most recent alternative and considers other paths (e.g., sub-expression "abab"). This process repeats until the machine either accepts the string in one path or rejects it after exploring all the alternatives without finding a match. This algorithm becomes extremely inefficient when the number of alternative paths grows exponentially. Consider the case of RE-based Denial of Services [315], where we may need to match the string "aaaa" and the RE "a?a?a?a?aaaa." Indeed, the backtracking approach has a time complexity of $O(2^m)$, where m is the number of alternative paths to be evaluated [49].

Thompson observed that the backtracking algorithms are mostly inefficient because they need to scan the input string multiple times [313]. To avoid this, he built a Virtual Machine (VM), implementing a multi-threaded execution model. The VM can handle simple operations like scheduling a thread, executing a thread for a specific time quantum, or a finite number of steps. Each thread executes code to match a single RE expression or sub-expression. Whenever a parallel or non-deterministic path occurs, it spawns additional threads to explore the alternatives with a breadth-first approach. In this way, the new threads do not require analyzing parts of the string already elaborated again. Moreover, we can avoid saving the whole *thread context* by executing the threads in "lockstep": all of them process the same character of the string and then move forward to the next [310].
Example: When applied to our example with the input string "abbb", Thompson's algorithm creates two threads for the sub-expressions "abbb" and "abab", respectively. All threads process the same input character in parallel, so they do not need to look backward in the string. After processing the first two characters 'a' and 'b', the third character of the input string is 'b', while the second thread is expecting the character 'a'. So it fails the matching and stops. The other thread can continue, consumes the remaining characters, and accepts the string. This approach offers an execution time that grows linearly with the number of string characters, while the degree of alternatives in the RE impacts the number of running threads per character and their relative cost.

5.2 Related Work

Previous researches explore either the algorithmic part of RE matching [313] or the efficiency of the execution platform [300]. The RE matching procedure usually executes through an approach based on DFA or NFA and their state transition table. While DFA suffers from the exponential memory footprint explosion [299,305], NFA requires high bandwidth to execute in parallel all possible active states [316]. Other solutions mix DFA-NFA characteristics to achieve the best complexity from both time and space requirements [298]. An example CPU-based engine is Intel Hyperscan [317], which overcomes main deep-packet inspection limitations with a novel regex decomposition mechanism and a CPU SIMD pattern matching for multi-string divided in a shift-or part and a verification of the false-positive part, at the cost of high preprocessing mechanisms. Nonetheless, modern automata processing benchmarks, such as ANMLZoo [290] and AutomataZoo [283], allowed designers to demonstrate the efficiency of FPGA implementations over general-purpose processors, spatial architectures like the AP, and GPUs [316].

A possible taxonomy of hardware execution engines considers the architectural type (Stream-Dataflow Architectures (SDA), In-Memory Architectures (IMA), Software-Programmable Architectures (SPA)), and approach/execution mode at their foundation (DFA, NFA, Hybrid, others), which are two orthogonal features. SDAs usually exploit spatial architectures, particularly reconfigurable ones, and focus on efficiently represent the considered (set of) Automaton(a). IMAs exploit the high bandwidth of their architectural design to perform an efficient lookup process of the Automaton(a) transition table. SPAs exploit a particular architectural style (even an IMA or an SDA) in a software-programmable way instead of regenerating a configuration bitstream with a time-consuming procedure. This Section gives an overview of these approaches and presents some state-of-the-art techniques applied in pattern matching engines, focusing on FPGAs. It will review the approaches starting from the execution mode features.

DFA hardware approaches, though they achieve better time complexity, generally suffer the memory footprint explosion for state transition table representation [303]. For this reason, many approaches focus on compression techniques such as states and transition clustering [305]. BFSM [299] encodes state transitions as rules, and groups similar transitions into a rule, achieving a very short and predictable memory lookup latency on an IBM PEN. Differently, Gogte et al. [284] implement a solution in ASIC, based

on the Aho-Corasick algorithm, overcoming its high memory requirements by splitting each input character in single bits. Tang et al. [318] propose a flexible real-time update Finite State Machine (FSM) and optimized DFA encoding scheme sacrificing the performance. Moreover, DFA's original execution scheme is limited to single character analysis. Hence, to tackle this issue, PiDFA [319] parallelizes character processing with the first computation of all the possible input character transitions, which is then merged in a pipelined fashion. Meiners et al. [285] propose a solution based on TCAM, which encodes the transitions of the DFA to improve the lookup process. Another body of work leverages hardware parallelism to increase the number of REs concurrently analyzed. Vasiliadis et al. [294] leverage the characteristics of GPUs to match multiple REs in parallel, encoding each RE as a separate DFA. Other solutions, like Sitaridi et al. [320], exploit GPUs to analyze known string matching algorithms, or embed in the FPGA logic many REs for Network-IDS [321].

NFA-based hardware approaches start with Sidhu et al. [302], that were the first to implement an NFA embedded in the FPGA logic, showing a flexible self-reconfigurable device, paving the way to further solutions directly synthesizing NFA [322] or exploiting dynamic reconfiguration [323]. Different solutions exploit either a partially reconfigurable solution based on a multi-character NFA [324] or a fully flexible structure with some patterns expression limitations [325] trying to overcome the run-time adaptability of approaches based on the fabric embedding. Other FPGA-centric approaches focus on specific applications, such as database query with fixed parametrizable operation [292,293] and signature-detection with YARA rules [287], which achieve remarkable performance results while sacrificing the time required for unknown REs at compile time. Instead, REAPR [159] is a tool that translates NFA into RTL implementations. The authors expand their work to support AWS F1 instances [36] and to allow a fast reconfiguration of different REs that exploit the same NFA structure [304]. On top of these approaches, other authors propose a compiler framework for spatial architectures called FlexAmata that aims at optimizing the automata representation also considering different alphabet symbols bitwidth [160], further extended to exploit either LUT- or BRAM-based designs [326]. The AP was an outstanding spatial reconfigurable architecture that embedded a target automaton into the reconfigurable fabric [300, 327, 328]. While it was a promising solution with high performance [160, 291], only simulation results of the AP were reported [304][3]. Nourian et al. [316] provide a comprehensive analysis of NFA solutions

[3] At the time of writing, the AP SDK is not available anymore

for different platforms (e.g., GPUs, FPGAs, Micron's AP) with different workloads and a partitioning scheme aimed at large NFA handling, showing that NFA-parallelism is not well suited to GPUs or vector architectures. In contrast, reconfigurable architectures are the most promising ones.

Hybrid approaches mix DFA and NFA to tackle their individual disadvantages. For example, Atasu et al. [306] use NFA to tackle traditional DFA limitations like repeating the matching process for each possible initial character or verifying an unbounded number of possible initial positions. It activates a new DFA for each first matching character, which significantly improves the performance at the cost of not scaling to complex or multiple RE. Others present a hybrid approach called Decomposed automata for efficient FPGA on-chip memories usage [329], or exploit multiple DFA based engines, called B-FSM [299], for a flexible NFA-like approach in an hetereogenous system [330], or build unique FA for wildcard pattern matching on TCAMs [331].

Different approaches do not rely on the use of Finite Automata but focus on achieving an efficient lookup process. For example, Agarwal et al. [332] propose a hash-based encoding scheme for text patterns (not specifically REs) that generates a dictionary matching engine. Instead, Nguyen et al. [333] employ bitmap index structures to encode the strings to match against, achieving multi-character lookups on FPGA.

Other methodologies consider REs as instructions, such as Google's library RE2 [334], which is mainly based on the Thompson's NFA and detailed by Russ Cox [310], exploits the *VM approach* (Section 5.1). Similarly, ReCPU [308] explores RE matching by translating a RE into a set of instructions executed on a "dedicated CPU" offering the run-time adaptability of a CPU with the performance efficiency of an ASIC solution, or on an FPGA [309]. However, its application to real scenarios is limited, and the absence of a communication subsystem prevents its adoption.

Employing REs as instructions provides an attractive alternative methodology to the automata embedding for building an efficient accelerator that can execute new "programs" without necessarily regenerating the hardware. Moreover, this methodology avoids the employment of explicit automata transition tables by exploiting the REs declarative language, similar to a Domain-Specific Language (DSL). This Chapter, and the following one, take inspiration from these approaches, seeing REs as a programming language, where each RE is a set of instructions repeated over a set of characters. Thanks this approach these Chapters present two DSRAs based on the two prominent execution modes (i.e., DFA and NFA) for the REs domain. These architectures rely on a custom compiler and ISA to represent

REs belonging to Regular Languages and on optimized microarchitecture and multi-core/-engine designs. In this way, these DSRAs deliver remarkable performance at higher energy efficiency. This Chapter mainly focuses on a DSRA based on a DFA execution mode (i.e., depth-first), showcasing its strengths and limitations. Finally, Table 5.2 summarizes the presented state-of-the-art approaches.

Table 5.2: *Related Work Summary based on platform, architectural type, compilation framework availability, and compilation time required (i.e., if closer to the software time or to the bitstream/hardware regeneration time)*

Work	Architecture Type	Execution Mode	Compilation Framework	Compilation Time Required
TiReX-FPGA	SPA	DFA	Yes	SW like
CICERO-FPGA	SPA	NFA	Yes	SW like
FPGA [309]	SPA	DFA	Yes	SW like
ASIC [308]	SPA	DFA	Yes	SW like
CPU [335]	SPA	DFA	Yes	SW like
CPU [317]	SPA	NFA	Yes	SW like
CPU [334]	SPA	NFA	Yes	SW like
GPU [294]	SPA	DFA	Yes	SW like
GPU [320]	SPA	DFA	Yes	SW like
FPGA [305]	SDA	DFA	No	HW like
FPGA [319]	SDA	DFA	No	HW like
FPGA/ASIC [284]	SDA	DFA	No	HW like/N.A.
FPGA [159]	SDA	NFA	Yes	HW like
FPGA [304]	SDA	NFA	Yes	HW like
FPGA [160, 326]	SDA	NFA	Yes	HW like
AP [160, 290, 304, 327]	IMA	NFA	Yes	HW like
FPGA [306]	IMA	Hybrid	No	HW like
FPGA [329]	IMA	Hybrid	No	HW like
Heterogeneous [330]	IMA	Hybrid	Yes	SW like
FPGA [331]	IMA	Hybrid	N.A.	HW like
FPGA [332]	IMA	Other	No	HW like
FPGA [333]	IMA	Other	No	HW like
FPGA [318]	SPA/IMA	DFA	N.A.	SW like
TCAM [285]	SPA/IMA	DFA	N.A.	SW like
PEN [299, 336]	SPA/IMA	DFA	Yes	SW like
FPGA [48]	SPA	DFA	Yes	SW like

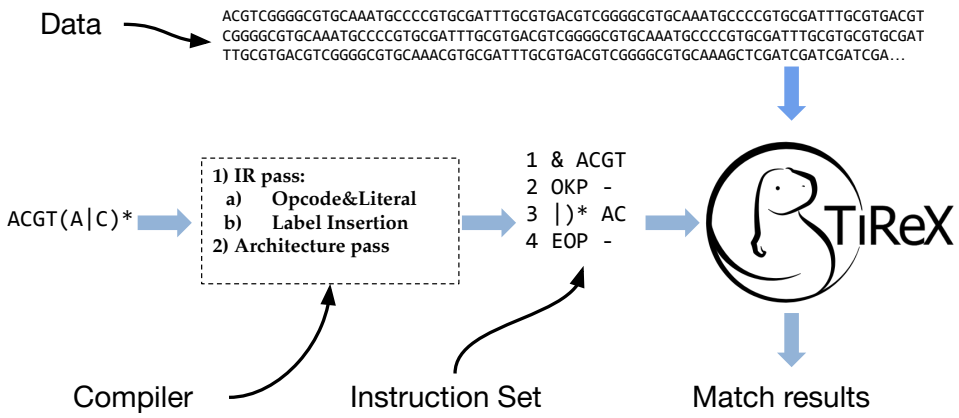


Figure 5.3: *TiReX flow for a Regular Expression.*

5.3 Design Methodology and Approach

This research work deals with REs as a programming language where REs are a sequence of operations repeated over a set of characters. The RE translates into a custom-defined ISA, on top of which we build our custom DSA called TiReX. The current ISA does not comprise advanced primitives of pattern matching that do not belong to Regular Languages or are derivable from the ones considered. Our DSA is not a general-purpose engine but is a software-programmable architecture tailored to the REs domain.

Figure 5.3 represents the workflow for executing RE matching on our DSA. The user provides both a RE (or a set of REs), the "program" TiReX will execute, and the data to analyze. The TiReX compiler translates the input RE(s) down to our low-level representation, i.e., the ISA. Then, the DSA takes as input the compiled code, which is loaded in the instruction memory. Finally, the architecture retrieves the data to analyze and outputs the matching procedure results. The next Sections first describe the compiler, then the ISA (Section 5.3.2), and finally the single- and multi-core architecture (Section 5.3.3 and Section 5.3.5), along with a performance model (Section 5.3.6).

5.3.1 Regular Expression Compiler

The compiler is a custom Python-based component that performs RE-to-ISA transformation. The compiler follows a Very-Long-Instruction-Word (VLIW) approach, where it is aware from the beginning of the underlining architecture characteristics and it produces the binary according to the

hardware features, e.g., the reference size determined by Execute parameters (Section 5.3.3.2). The compiler builds on two main passes: the Intermediate Representation (IR) pass and the architecture-aware pass. On the one hand, the IR level pass divides into two phases. The first one performs opcode and literal recognition and detects the need for a second pass for jump address insertion. The second (and optional) pass deals with labeled locations from the first pass and inserts the addresses for absolute jumps. On the other, the architecture-aware pass accounts for character parallelism abilities and word alignments.

5.3.2 Instruction Set Architecture

Table 5.3: *Opcode encoding of the operations.*

opcode	*RE*	*Description*
000000	EOP	End of Pattern
010000	AND/&	And of cluster matches
001000	OR/|	Or of cluster matches
011000	.	Match any character
100000	(	Function call/sub-RE start
000100	)	Function return/sub-RE end
000001	$)^*$	Match any number of sub-RE
000010	$)^+$	Match one or more sub-RE
000011	)|	Match previous sub-RE or next one
000101	OKP	Open Kleene Parenthesis
000111	JIM	Jump If Match

The TiReX ISA approach relies on a VLIW-like machine, where multiple operations are bound together to form a bundle. Table 5.3 shows the primary operations that compose the TiReX ISA primitives. These primitives are composable operators to form different REs, with their literal part, and are at the basis of all the complex REs, such as repeating n times the string x, i.e., $x^{\{n\}}$. The compiler decomposes advanced REs into TiReX primitives. Complex REs carry the cost of a bigger program size; therefore, the instruction memory has to accommodate a reasonable amount of instructions or to provide an adequate memory hierarchy. For example, the RE $a^{\{3\}}$ translates into a sequence of *aaa*, hence turned into a simple concatenation of three *'a'* characters. Each instruction is divided into two fields: the opcode field, where we encode different pluggable operations, and the remaining part, which is called reference or instruction operands. The reference field usually contains the number of parallel characters a TiReX core can crunch, or, in case of particular operators, it contains help-

ful information such as the branch target address. In our implementation, the instruction word is *38-bit* long, and it has *6 bits* for representing the opcode field, while the remaining *32 bits* represent the reference field, i.e., at most four parallel standard ASCI characters. TiReX operations divide into three main groups: Character Match, Control Flow, and Support.

Character Match Operators represent the basic operations on REs to match characters with boolean logic, like the AND instruction to match a fixed sequence of characters or the OR instruction to accept several alternative characters.

Control Flow Operators represent an advanced class of operators to compose advanced REs against the Character Match operators. In particular, these operators control the flow of the instructions (e.g., jump), perform a "function" call preserving the "context" (e.g., matching process status, current data and instruction address, prefetching hints). These operators are described as follows. The **EOP operator** is a special instruction whose purpose is to signal the end of the program (i.e., the end of the RE matching procedure). The **open parenthesis or) operator** represents a "function call", or sub-RE, and translates to context preservation. **The closed parenthesis or) operator** represents a "function return"; therefore, it instructs the processor to restore the previous context. The $)^*$ **and** $)^+$ **operators** represent the Kleene operators, at the end of a function call, which, like a for-loop, re-execute the loop body of the matching code until either a mismatch occurs or the string ends. The $)|$ **operator** composes chains of OR-ed REs: the sequence of functions matches when any of the REs matches, in which case the hardware can skip the remaining functions in the chain.

Support Operators are instructions to support the Control Flow ones and increase performance. Indeed, they are needed to inform the architecture of the instruction memory location to jump. In particular, the function call operator "(" can ambiguously lead to a function with the Kleene operators or an OR chain: in the former case, the architecture has to re-execute the function, hence jumping to a previous instruction, while, in the latter case, the flow of instructions goes forward and the architecture can skip the following REs in case of a match. For this reason, we specialize the "(" operator to discriminate the two cases and inform the architecture prefetchers on which instructions and data TiReX has to load in the next cycle for all the possible comparisons results. The **OKP operator** hints the architecture about the presence of a for-loop-like sequence, hence showing a possible backward jump in a matching case or a forward jump in a mismatch case. The **JIM operator** gives the flow-controller a hint of the presence of OR-chained function calls, therefore preparing for a possible forward jump in

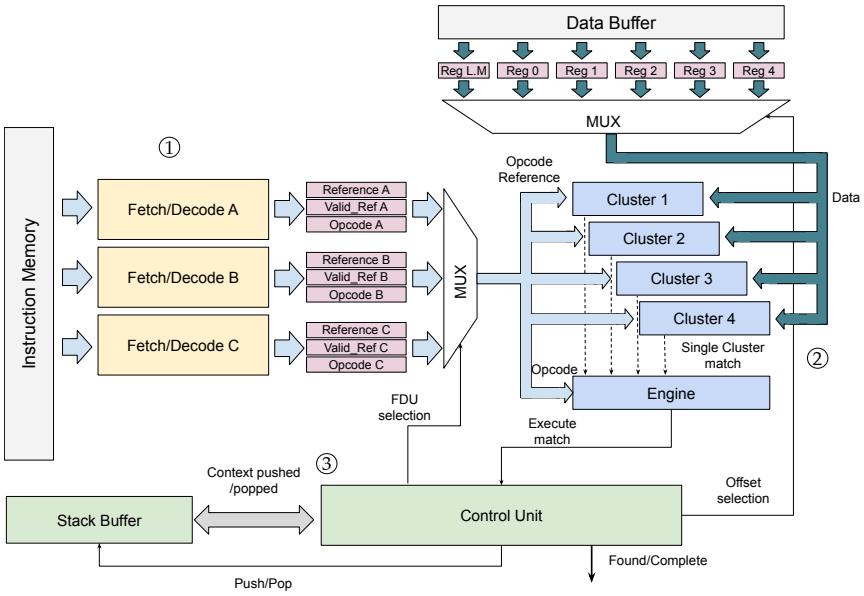


Figure 5.4: *Details of the core logic, with the pipeline components and the reference to Section 5.3.3 subsections.*

the flow whenever match. In contrast, a mismatch causes the architecture to load the next instruction and re-read previous data (which can be cached).

5.3.3 Single-Core Architecture

TiReX DSA has a two-stage pipeline divided into Fetch/Decode and Execute stages. The core tracks the RE procedure status, and it can be in two different states (i.e., match or not match state), which determine different execution flows. Figure 5.4 shows the block design of the microarchitecture. The main components of the core are the Instruction Memory (IM), which stores the program instructions, the Fetch/Decode Units (FDUs) for the homonymous stage, the Data Buffer (DB), which stores a portion of the input data necessary for the matching process, the Clusters and the Engine, which form the Execute Unit (EU), and the Control Unit (CU), devoted to the control of the whole matching process.

A general flow of the matching process starts with the loading of the compiled RE into the IM, then the FDU loads one instruction at a time, decodes it, and propagates the control signals (e.g., the *valid_ref* signal, which indicates how many valid characters are in the reference in one-hot encoding scheme) and the instruction operands to the EU and the CU. The EU loads the input characters from the DB and searches for patterns accord-

ing to what received from the FDU, emitting a match to the CU when the current input characters match the pattern encoded in the current instruction. Finally, the CU exposes signals to the external logic to indicate the completion of the matching process and the presence of a match. The basic RE matching is an intrinsically sequential control dominated flow that analyzes a single character per clock cycle [44]. We now describe how TiReX deals with control hazards within a RE instruction flow (Section 5.3.3.1) and how we increase the number of characters analyzed in a single clock cycle (Section 5.3.3.2) showing remarkable performance (Section 5.5.3.1).

5.3.3.1 Fetch/Decode stage

The Fetch/Decode stage consists of three copies of an FDU, exploited to prefetch every possible instruction flow. An FDU takes the bundled word from the IM and unpacks it in the three pieces of information needed for the RE matching procedure: the opcode, the reference, and how many references are really in that field, i.e., the signal *valid_ref*, as shown in Figure 5.4. Thanks to the Control Flow Operators, which we tailored to change the instruction flow, we exploit different instruction-pre-fetching mechanisms. By instantiating three different and specialized FDUs (marked A to C in Figure 5.4), the core can avoid cycle losses in the case of *not match* or case of special instructions such as the JIM or the OKP. The identified flows are mainly three.

Sequential Execution: The RE matching process can run in the simple sequential execution flow. Indeed, the FDU-B continuously prefetches the next instruction, essential when a match is found. Thus, we cover the basic sequential execution flow of a "program".

Instruction Rollback: The "program" can find a false initial match up to a certain point. Indeed, discovering a false partial submatch requires to rollback the execution to the first "program" instruction. Since restoring everything in case of a false match leads to large cycle loss, FDU-A keeps a copy of the very first instruction and its control signals.

Special Jump: Another possible performance degradation source comes from jump instructions. Indeed, as in general-purpose processor, control flow modifications that depend on run-time computations, such as a comparison result, require special hardware components (e.g., branch resolution anticipation, dynamic branch predictors) or clock cycle stalls for control resolutions. Jumping back and forward in the "program" leads to control hazards. Considering a backward jump, as for the Kleene operations, or a forward jump, as for OR-chain, we need to know the target instruction.

To handle this, the second pass of the compiler hints at the core by inserting this target jumping address. The FDU-C leverages compiler hints to prefetch instruction located after the sequence of OR-ed REs (in case of JIM) or the initial instruction of a sub-RE (in case of an OKP).

5.3.3.2 Execute stage

The other datapath portion is the EU, and it consists of two parts: the Clusters and the Engine.. The Clusters are the components that compare input data against input reference; hence, the more the Clusters, the more the characters the core can analyze. The most basic primitive in the RE matching process is a comparator for single character-to-character comparison. More "advanced" primitives are *AND*, or concatenation of characters, and *OR*, or alternation that in our ISA corresponds to the Character Match operations. These primitives correspond to TiReX Clusters, and each of them takes as input the reference characters and the data characters. The EU has two orthogonal design parameters that determine the parallelism degree in the character analyzable per clock cycle. The first one is the width of the Clusters, called *ClusterWidth* parameter. Figure 5.5 shows a ClusterWidth equal to 4. Hence, that Cluster performs at most four character comparisons in a clock cycle. Depending on the opcode, we feed the Cluster with different data characters. In the case of an *AND*, the four comparators will receive different characters, as in Figure 5.5. In the other case, i.e., the *OR*, each comparator will receive the first data character of the substring (e.g., the first C in the CCGT substring), and compare against all the reference characters. In this way, the ClusterWidth determines the maximum amount of characters the core can analyze with the *AND* opcode.

The other design parameter regards how many Clusters the EU has, called *NCluster*. Considering Figure 5.5, NCluster is equal to four. Following the previous data feeding scheme, each Cluster will receive a substring shifted by one of the data characters, i.e., Cluster one will receive CCGT, Cluster two CGTA, etc. Hence, considering an *OR*, we feed each Cluster with a single data character, while, with *AND*, we feed them with four characters. The Clusters then compare such characters against the reference ones. In this way, the NCluster parameter determines the maximum number of characters the core analyzes with an *OR* opcode.

The Engine collects all the intermediate results from the Clusters, knows the current Character Match operation, and combines them to produce an aggregate result for the Control Unit. The Engine works in two different ways, depending on being in matching or not matching state, and it controls

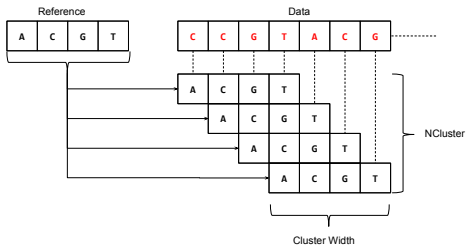


Figure 5.5: *Detailed view of the EU during the first AND instruction execution.*

Instruction	Cycles							EU Data
	#1	#2	#3	#4	#5	#6	#7	
& ACGT	FD	EX						CCGTACG
& ACGT		FD	EX					ACGTATT
OKP			FD	EX				ATTGCAC
)* AC				FD	EX			ATTGCAC
)* AC					FD	EX		TTGCACT
EOP						FD	EX	TTGCACT
								Match found
Instruction:	*ACGT(A|C)**							
Data:	*CCGTACGTATTGCACTA*							

Table 5.4: *Execution example of a Regular Expression on a single core (the effective data analyzed are underlined).*

which Cluster is active or not, stalling the execution if necessary. All these components are almost combinational; hence, balancing the EU parallelism is crucial to avoid unused or small resources.

Overall, the architecture might have different inputs feeding schemes depending on the instruction result (*match/not match*) and the number of matching characters (from 1 to ClusterWidth). For this reason, we adopt different data-shifting mechanisms for the DB, where we use registers to store possible inputs for the next instruction.

5.3.3.3 Control stage

While the EU handles the Character Match operations, the CU handles all the other complex operations, e.g., Kleene ones. Indeed, this unit is a centralized controller that synchronizes the different prefetching and prediction mechanisms. It is aware of the current RE matching procedure status, e.g., match or not match, enabling dynamic scheduling of the proper instruction, keeps track of the instruction and data pointers, and generates the proper addresses for the memories. Moreover, whenever the Control Unit finds a function call, e.g., the "(" operation, it pushes the current RE matching context to the Stack Buffer, our context memory, and synchronizes the datapath to work in this new context.

5.3.4 Regular Expression analysis example

Table 5.4 shows an example of the matching process on the TiReX core. First, the FDU-A retrieves the very first instruction producing the *AND* opcode, ACGT as reference, and *valid_ref* signal equal to 1111, since every character in the reference is valid. Then, the EU performs the comparison, resulting in a mismatch. Indeed, as shown in Figure 5.5, the reference is matched against the input stream in NCluster exact starting positions, given

the initial not-matching state. Thanks to the presence of multiple FDUs, the CU does not flush the pipeline as the first instruction is still present in FDU-A. The data *pointer* jumps four characters ahead (due to the presence of four Clusters), and the next batch of characters is checked against the first instruction. This process is executed until a valid intermediate match is found (this happens at cycle *#3* in the example), which moves the CU to a matching state and increments the program counter and data pointer by one and four, respectively. This offset is computed from the Engine and depends on which is the matching Cluster. The core continues until it reaches the last instruction (EOP) and reports the final result, in this case a match found.

5.3.5 Multi-Core Architecture

The proposed single-core DSA adopts improved parallelism at the character level. Although we could consider it similar to a SIMD architecture, the domain requirements (i.e., real-time analysis and flexibility) are not yet satisfied. Hence, we propose a multi-core architecture based on the replica of the same tile described previously. Each tile has its private memories, separate for data and instruction, and it is software programmable to work in two different ways depending on the needed parallelism level, as shown in Figure 5.6. According to the application, we tailor the system to the RE recognition process, which can operate in two different modalities, i.e., MISD- and SIMD- like.

Multiple REs Single Data Stream. Considering a scenario where there is a need to analyze a wide range of patterns simultaneously, e.g., Signature-Based Detection [289], we designed a dedicated multi-core architecture based on the TiReX tile described previously. Each core is equipped with its private instruction memory, i.e., different REs that it has to deal with, while the data stream is the same for every core. In this way, we can increase the number of patterns processed per single execution while keeping a similar data crunching ability. However, this execution mode requires further investigation in the topology of the architecture, in the interconnection logic, and can lead to data divergence and issues related to memory coherency when considering terabytes of data, such as in the genomic case. We will not address this issue here because we think it is out of the scope of this work, and we think that exploiting memory coherent protocols (e.g., OpenCAPI [145], CCIX [146], or CXL [337]) is a sufficient solution.

Single RE Multiple Data Stream. In contrast to the previous version, this operational way aims to exploit the parallel core to increase data

crunching rates. Indeed, a fast scan rate is essential when considering a vast amount of data to analyze, such as the Human Genome, or big database. To improve our single tile abilities, we fill each tile IM with the same RE while providing different data stream portions. These chunks of data are not independent a priori, and potential matches in the crossing regions may happen. Therefore, we do not consider a sharpened cut, but we adopt a simple heuristic to provide an overlapping region to avoid false mismatch at a negligible overhead cost. The data splitting task is currently handled at the software level and relies on a domain-expert specified threshold that indicates the maximum length of a match, translating into the overlapping region. We compute the *batch size*, or B_{size}, for each of the *i-th* core of the *N* ones, through $B_{size} = \frac{S_{data}}{N_{tc}}$, which divides the *size of the data* (S_{data}) by the *number of cores* instantiated in the system (N_{tc}). With the batch size, we compute the *End of Data*, or EoD_i, $\forall i \, \epsilon \, N \; EoD_i = min(B_{size} \cdot (i+1) + Tr, S_{data})$ per each *i-th* core by simply adding a user-defined *threshold* (*Tr*) to ensure coverage of a possible matching sequence of *Tr* maximum length. Finally, we compute the new *Start of Data*, or SoD_i, except for the first core, which computes from the very beginning, $\forall i \, \epsilon \, N - \{0\} \quad SoD_i = EoD_{i-1} - Tr$, as the difference from the previous core ending point minus the domain-specific threshold, ensuring coverage of a possible matching sequence of *Tr* maximum length.

5.3.6 Performance Analysis

Here follows a theoretical performance model to analyze the expected throughput in both the worst and the best cases. Although the matching process is highly data-dependent, the time required to process a single character is the main critical parameter measured in $\frac{ns}{char}$. This *time-per-char* metric (the lower, the better) is mainly affected by the process status of the RE, i.e., match or not match case. Being in a not matching case, the time per char T_{nm} depends on the system frequency, *F*, and the number of characters processed per clock cycle. In the worst case, it is *NCluster*; in the best case, it implies the full utilization of the Execute unit. Consequently, $T_{nm} = \frac{1/F}{NCluster}$ and $T_{nm} = \frac{1/F}{NCluster + ClusterWidth - 1}$. On the other hand, considering a matching state, we span from a single character per clock cycle in case of union, or OR, $T_{mu} = \frac{1}{F}$, to a number of *ClusterWidth* character processed in the concatenation case, or AND, $T_{mc} = \frac{1/F}{ClusterWidth}$. Substituting the effective design parameters, we can compute the worst and the best case of the time per char required to analyze a character at the single-core architecture level. Additionally, we can model the expected through-

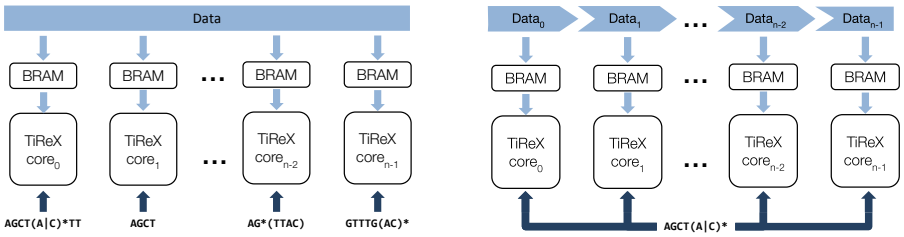


Figure 5.6: *Multi-core modalities with multi REs single data stream, or single RE multiple data streams, respectively.*

put of the architecture, both single- and multi-core, adopting the presented equations and estimate the *bitrate* of the system in *Gb/s* as $B_x = \frac{1}{T_x} \cdot 8 \cdot N_{tc}$, where T_x is the time per char obtained with the previous equations, N_{tc} is the number of TiReX cores.

These equations provide an estimate of bitrate values for possible implementations of TiReX with a variable number of cores and frequency, enabling a fast performance scaling. This makes it possible to estimate the goodness of the various TiReX implementations before going through empirical measurements.

5.3.7 Architecture analysis summary

To summarize, the proposed architecture relies on a custom-developed ISA for Regular Expression matching flexibly. In contrast with many of the NFA/DFA approaches, the proposed methodology handles different runtime tunable REs without modifying the underlying architecture. The current execution model of the architecture is based on a depth-first-like approach, where the execution of the instructions is intrinsically sequential [44]. Though it suffers from classical backtracking issues, this model could lead to possible future solutions based on a breadth-first-like model, similar to a theoretical NFA, achievable through mainly NFA logic embedding. Additionally, the architecture has two levels of parallelism to increase execution efficiency. The first level of parallelism resides in the number of characters analyzed per clock cycle, which is a design parameter, and all the prefetching mechanisms to handle multi-flow executions. The second level of parallelism relies on a private memory multi-core architecture that can employ two different theoretical execution models, i.e., MISD- and SIMD- like. The first model increases the number of parallel REs analyzed, while the second one increases the parallelization of vast amounts of data.

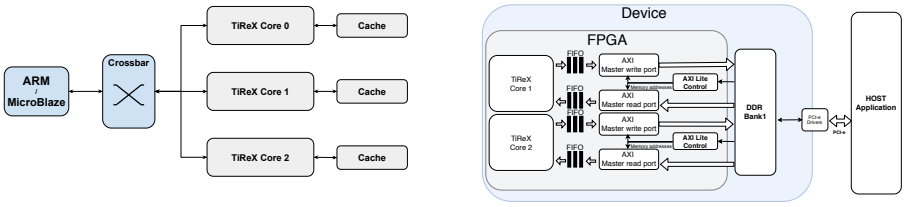


Figure 5.7: *Architectural model of the embedded and external host implementation of the TiReX system.*

5.4 Scenario-Specific Architectural Models

To demonstrate the adaptability of this work, we implement the system following two different architectural models. The first model targets all those platforms in embedded-like scenarios, e.g., a System on a Chip (SoC) comprising a CPU and an FPGA. This model targets latency as the primary metric and tries to increase energy efficiency as much as possible. This specific architecture is called *Embedded Host*, given that the host application executes on a CPU tightly coupled to the FPGA. Instead, the second model involves server-like systems targeting a throughput-oriented scenario, where tackling highly intensive data workloads is paramount. In this model, called *External Host*, the host application executes on the server processor connected through a PCI-e bus with the accelerating device.

5.4.1 Embedded Host

This model targets a more constrained execution environment where the FPGA is tightly coupled with a CPU. Examples are the PYNQ platform provided with ZYNQ technology, which embeds an ARM Cortex A-9, or a softcore, such as the MicroBlaze, directly instantiated on the programmable logic. The embedded processor and TiReX communicate through lightweight AXI-Lite ports for fast, small transactions of both control or data of 32 bits per transaction. We instantiate a multi-core architecture interconnected through a crossbar, as in Figure 5.7, by a simple tile replica, each of which has its additional private cache memory, implemented through BRAMs, to exploit the available reconfigurable fabric fully.

The overall system starts with the processor loading the various caches and instruction memories of all the TiReX cores instantiated in the system following a SIMD or MISD fashion depending on the user's needs. Once filled, the processor enables the beginning of TiReX cores computation. Once one of the cores produces a result, the search completes, reporting

the outcome to the host. Section 5.5.3.1 shows the results this architectural model can achieve in latency-sensitive scenarios.

5.4.2 External Host

Thanks to the presence of cloud FPGAs, everyone can access a high-end cloud FPGA that communicates through a peripheral bus to an external host processor. We devised the external host model to target such a scenario. Consequently, we can implement TiReX on systems like the AWS F1 instances, which contain high-end FPGAs equipped with four physical DDR ports. Each physical port can transmit up to *512 bits* of data per clock cycle, for a total of *2048 bits*. Moreover, we can time-multiplex each physical port for logic port interleaving and instantiate a wider number of cores. Therefore, we decide to exploit this chance by deploying the kernel via the SDAccel framework, as shown in Figure 5.7. The tool automates some system design steps and provides a communication infrastructure, which requires specific design interfaces and APIs for the transmission phase among host and device through the high-bandwidth PCI-e link. Each tile has an AXI-Lite interface for control exchange, while most of the data exchange goes through an AXI-Master port attached to the DDR bank. The interface requirements need additional *glue logic* and FSMs to handle instruction and data transmission, and some performance counters. The Master port passes the data through a FIFO to provide a back-pressure mechanism for DDR-tile exchange, and the same happens in the other direction. We replicate the tile and the additional interfaces to provide a multi-core architecture, where each core is independent.

The final system flow starts with the RE(s) compilation. We feed the data according to the operational way and apply the splitting heuristic if needed. Then, the host writes the data to the DDR through the PCI-e and starts the FPGA kernel. Each core, which has a private portion of the assigned DDR bank, reads and loads the instructions and then retrieves the data to analyze. Once done, the overall kernel writes back the computation results, while the host waits for all the cores to end and reads the match outcome along with performance counters, such as clock cycle count and matching position. Section 5.5.3.2 details the performance results the External host architectural model achieves in throughput-oriented scenarios.

5.5 Experimental Setup and Results

We designed TiReX architecture in VHDL, with additional glue logic in System Verilog. We targeted three different Xilinx platforms: the VC707

and the PYNQ-Z1 boards with an embedded host model, while the external host model makes use of the Amazon Web Services (AWS) F1 instance powered by a Virtex UltraScale+ 9 (VU9P) FPGA attached through a PCIe connection, paving the way to deploy TiReX-as-a-Service. While the VU9P system is a more throughput-oriented solution, thus suggested for a server-like system use case, the PYNQ-Z1 system represents a more constrained use case, such as embedded systems for an autonomous vehicle or IoT scenarios. We use Xilinx Vivado and SDK 2016.4 to generate the bitstream and manage the bare-metal host for the embedded host model. For the external host model, we use Xilinx SDAccel 2016.4 to implement the system on the VU9P, and we exploit the OpenCL library for the host. We employ four different state-of-the-art software to compare against TiReX. The considered baseline is FLEX, which produces a DFA for each RE in a C file that needs to be compiled. Then, we employ Grep, which builds the matching NFA at run-time, Google's RE2 [334], an optimized multi-threading C++ library that builds an NFA as well, and Hyperscan [317] by Intel, which applies an intensive preprocessing mechanism to decompose the RE(s) into a SIMD NFA. We compile FLEX code, RE2 library, and Hyperscan with *-O3* optimization level and adapt the software tools to stop as soon as a match is found. In particular, for RE2 we use the `RE2::PartialMatch` function, while for Hyperscan we employ the `hs_compile` with the `HS_FLAG_SINGLEMATCH` flag. For reference CPU, we have the ARM Cortex A9 of the PYNQ-Z1 running at 650 MHz, an Intel i7-8750H with six cores and a peak frequency of 2.2 GHz, and a dual-socket Xeon E5-2680 v2 with a total of twenty cores and a peak frequency of 2.8 GHz.

We focused on bioinformatics and networking to test TiReX in real-life fields with state-of-the-art benchmarks. We divided the testbenches into two sets: a *small* one, which we indicated as *S*, aimed at a latency-oriented scenario, and a *large* one, told as *L*, targeting a throughput-oriented scenario. For the S scenario, we used the first *16 KB* of the first human chromosome [338]. We matched this input against the three REs, whose complexity increases to measure the internal core execution latency correctly and stress the performance. For the L scenario, we first selected the *E-coli* bacteria *proteome*, about *8.5 MB* of data retrieved from UniProtKB database [339], and used *PROSITE* [340] as the REs dataset. PROSITE consists of a biologically significant database and patterns formulated to reliably identify which known family of proteins the new sequence belongs to, including the REs. Then, we tested the throughput with the PowerEN benchmark from ANMLZoo [290] with 1MB of data, which

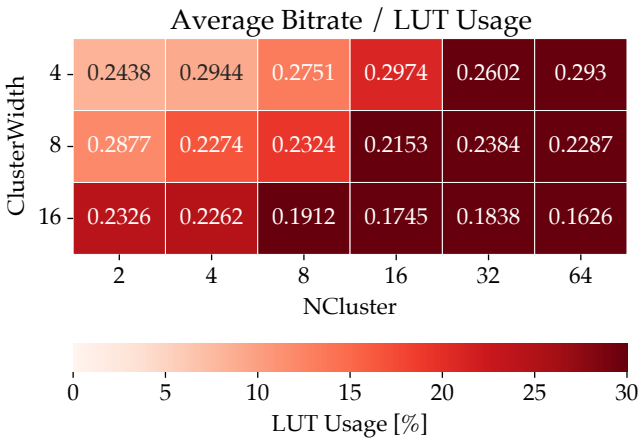


Figure 5.8: *Heatmap representing ClusterWidth and NCluster space exploration considering the ratio of average bitrate and LUTs usage on a PYNQ-Z1. The darker the color the closer (or beyond) the resource usage to the given budget for a TiReX single-core. The (4,4) configuration reports a good trade-off while respecting the budget.*

has been employed in the validation of the IBM PowerEN for networking function embedding at the edge. We analyze both the execution time and energy efficiency of the proposed architecture for these testbeds. We collect the power consumption of the whole board, hence including the host, through a Voltcraft 4000 energy logger for the VC707 and the PYNQ-Z1, while we take the Thermal Design Power (TDP) for the VU9P as from a literature work [341], i.e., 42W, excluding the host. Besides, we use the TDP for the Intel CPUs while we measure the ARM A9 power consumption using the energy logger. We compute the energy efficiency as $throughput/power\ consumption$, where throughput is the inverse execution times.

5.5.1 Exploration of Design Parameters

Here, we explore ClusterWidth and NCluster design parameters employing an open-source automation tool [53] and targeting a PYNQ-Z1 device. The exploration aims at finding a trade-off of these parameters that shows noteworthy performance while keeping a low critical resource footprint (i.e., LUTs). Figure 5.8 reports a heatmap with the ratio of average bitrate, and LUTs usage. We set a maximum budget of 30% of LUTs for a single core since we aim at scaling in the core number; hence, Figure 5.8 shows darker configurations that are close (or beyond) this upper bound. Among the

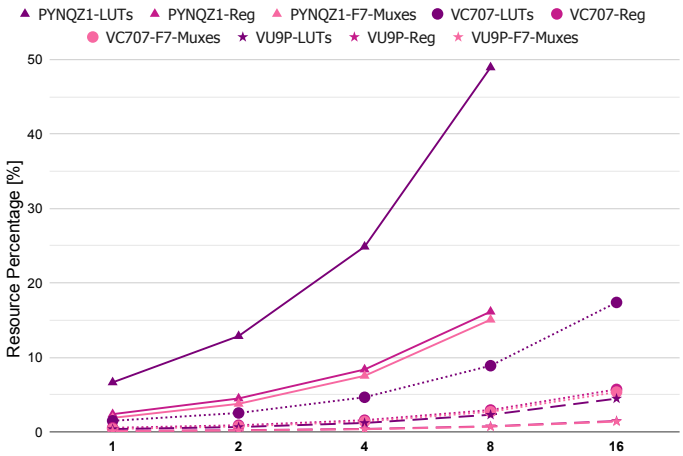


Figure 5.9: *Resources scaling (darker to lighter: LUT, Reg, Mux) against core number on the PYNQ-Z1 (triangles), VC707 (dots), and VU9P (stars).*

lighter configurations, (4,4) and (4,16) report remarkable performance, but the (4,4) configuration shows less resource usage. In this context, the (4,4) represents the optimal trade-off of delivered bitrate per employed LUTs that eases the core scaling even with resource-constrained devices.

5.5.2 Multi-core scaling synthesis results analysis

Here, we aim to exploit the reconfigurable fabric as much as we can to scale the core number and reach comparable state-of-the-art results. Figure 5.9 shows the resource scaling with power-of-two cores (though every number of cores is suitable) architectures; in this way, we explore a reduced portion of the design space and simplify the memory traffic model. Given the limited number of physical memory ports, those channels have to work in interleaved mode. However, increasing the number of nodes attached in an unbalanced way may not result in deterministic performance prediction and load management requirements. To devise a multi-core architecture, we first investigated the area utilization of a single core implementation, which can provide insights into the number of cores that can fit in a multi-core design. Figure 5.9 shows a small resource footprint of a single TiReX core on the three target platforms in all cases. We report the most relevant resources utilization, given the nenglibile amount of the others. Therefore, they do not represent an issue at all. Although the numbers in Figure 5.9 suggest that it is possible to deploy a high number of cores on each platform (like more than a hundred for the VC707), this is not possible due to

routing-related issues. Indeed, a high number of cores leads to congestion on the interconnection paths towards the processor and the communication infrastructure, with very high fan-outs hindering the routing phase. These issues limit the deployable cores to 16 on the VC707, given that designs beyond 32 cores fail during the routing phase. However, on the PYNQ platform, the number of deployable cores scales down to 8 since designs with more cores (from 16 on) or targeting higher frequencies fail during the routing phase, having fewer resources available than the VC707. We envision designing a dedicated interconnection network to scale to more cores while keeping acceptable frequencies and synthesis time. However, this is beyond the scope of this work.

Conversely, on the AWS F1 instance, the limit of 16 cores is not due solely to routing problems, even if the area utilization also comprises the more significant AXI logic. In this scenario, the limitation is due to the number of logic Advanced eXtensible Interface (AXI)-Master ports that can be instantiated. The toolchain constraints to up to 16 different AXI-Master logic ports. Indeed, with internal design unchanged, each core retrieves its data portion from the DDR banks after a setup phase from an external host through PCIe. To instantiate a higher number of cores, it is necessary to insert back-pressure mechanisms in the interconnection logic and the internal logic itself for handling variable unavailability of data. The data transfer rate is limited to *2048 bits* per clock cycle, since there are at most four-Double Data Rate Memory (DDR) ports available for each DDR memory bank, each one capable of transmitting *512 bits* per cycle.

5.5.3 Performance Analysis Against Software References

Alongside the synthesis results, we compare our system against state-of-the-art software approaches in terms of performance, accounting for the latency (*S* scenario) and the throughput (*L* scenario), and considering the energy efficiency. The following experiments evaluate the multi-core architecture presented in this paper in SIMD modality and the single-core architecture on mainly the VU9P device. We adopted the heuristic presented in Section 5.3.5 and a threshold *Tr* equal to 100, which we manually computed to be suitable for both the *S* and *L* scenarios. We analyze the data transfer overhead from the host processor to the target FPGA on the external host model, focusing on the L scenario. The overhead is generated both by hardware and software components. On the one hand, the data transfer occurs via a PCIe-Gen3x16 connection, which has a transfer rate of up to 15.6 GB/s, hence providing a latency based on the input data size.

Table 5.5: *Performance results (in bold the best) of the S scenario tests, chosen to stress incrementally our architecture (FLEX on the i7 is the baseline, while FLEX on the A9 is omitted being far slower than Grep on A9).*

Method	Architecture	Exec. Time [μs]			Speedup			Energy Efficiency [$1/(ms \cdot W)$]		
		Test 1†	Test 2‡	Test 3*	Test 1†	Test 2‡	Test 3*	Test 1†	Test 2‡	Test 3*
Grep	ARM A9 650 MHz	11963	12185	12374	0.02×	0.01×	0.02×	0.02	0.02	0.02
RE2	ARM A9 650 MHz	589	353	391	0.46×	0.34×	0.67×	0.42	0.70	0.64
FLEX	Intel i7 2.2 GHz	271	121	263	1×	1×	1×	0.08	0.18	0.08
Grep	Intel i7 2.2 GHz	492	805	221	0.55×	0.15×	1.18×	0.04	0.03	0.10
RE2	Intel i7 2.2 GHz	50.69	29.30	41.98	5.35×	4.13×	6.27×	0.44	0.76	0.53
Hyperscan	Intel i7 2.2 GHz	78.92	53.58	35.08	3.43×	2.25×	7.50×	0.28	0.41	0.63
FLEX	Xeon E5 2.8 GHz	598	136	404	0.45×	0.88×	0.65×	0.01	0.06	0.02
Grep	Xeon E5 2.8 GHz	205	108	336	1.32×	1.11×	0.78×	0.04	0.08	0.02
RE2	Xeon E5 2.8 GHz	34.48	23.02	28.28	7.86×	5.25×	9.30×	0.25	0.38	0.31
Hyperscan	Xeon E5 2.8 GHz	59.49	52.21	27.95	4.56×	2.32×	9.41×	0.11	0.17	0.33
TiReX	VU9P 1 core 299 MHz	37.66	18.32	29.63	7.19×	6.60×	8.87×	0.63	1.30	0.80
TiReX	PYNQ-Z1 8-core 70.5 MHz	7.20	8.21	30.30	37.63×	14.73×	8.67×	**41.75**	**36.61**	9.92
TiReX	VC707 16-core 130.1MHz	2.07	4.54	3.36	130.9×	26.65×	78.27×	22.74	10.37	**14.01**
TiReX	VU9P 16-core 202.7 MHz	1.03	0.75	2.96	**263.11**×	**161.33**×	**88.85**×	23.11	31.74	8.04

$^{\dagger}ACCGTGGA \quad ^{\ddagger}(TTTT)^{+}CT \quad ^{\star}(CAGT)|(GGGG)|(TTGG)TGCA(C|G)^{+}$

For example, the E-coli bacteria protein dataset size is *8.5 MB* and the resulting transmission time is around *544 μs*. On the other hand, the software introduces non-negligible overheads: the time for the call to the OpenCL Application Programming Interfaces (APIs), the CPU context switch time, the time required for the interrupt to be served, and the additional asynchronous driver time. The sum of the previous software-based overhead times and the system setup time for the execution of the first instruction is around *70 μs*. This value has been computed by injecting empty instructions and empty data into the core, eliminating the data transfer overhead. The transfer rate, along with the software-based overheads, provides the base execution times of the TiReX system that has to be taken into account.

5.5.3.1 The S Scenario

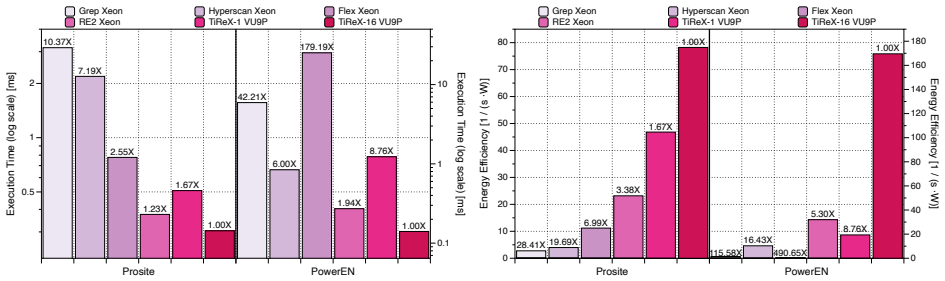
This scenario evaluates the latency of the considered approaches on the first *16 KB* of the first human chromosome [338]. We employ three different tests to stress the methodology effectiveness, from string matching to more complex RE matching tests. Table 5.5 summarizes the performance results of the S tests run on the various platforms and compared against FLEX, Grep, RE2, and Hyperscan without considering *data transfers* or *I/O parts* for all the considered methods. On the other hand, we account for the preprocessing mechanism of Hyperscan since this phase highly influences the matching methodology and performance. Moreover, we did not evaluate Hyperscan on the ARM A9 since it does not officially support ARM-based

architecture. Considering TiReX multi-core implementations, we achieve speedups ranging from a 0.92× up to 33.47× against the best software implementations of Hyperscan and RE2. The slowdown comes from the single-core on the VU9P and the PYNQ-Z1 with eight cores against RE2 and Hyperscan on Test 1 and 3. We must consider the differences in the execution model, our DFA-like versus software NFA-like, and the considered platforms. Indeed, both the software tools run on a server-class CPU, while the PYNQ-Z1 is an embedded device, and the running frequencies are incredibly different, i.e., 2.8 GHz versus 70 MHz. Hence, the achievement of performance in line with state-of-the-art tools on a server-class processor demonstrates the remarkable benefits of our approach even when employing an embedded device. Moreover, scaling cores on the VU9P showcases a considerable improvement (i.e., a top of ∼69×) against the state-of-the-art software solutions, validating the proposed multi-core approach.

Considering the energy efficiency in Table 5.5, or ratio of throughput over power consumption, TiReX DSA provides a higher degree of efficiency than software solutions. In particular, TiReX multi-core delivers a remarkable energy efficiency spreading from about 12× up to 95× against the most efficient CPU implementations. Thanks to these tests, we demonstrate TiReX advantages of specialized hardware for both execution times and energy efficiency starting from the single-core to the highly parallel multi-core implementations. Although the 8-core PYNQ-Z1 implementation achieves the same speedup as the VU9P single-core one, we should consider the working frequency's role in these results. Indeed, the two running frequencies differ by 220MHz, which gives a non-negligible lead. Additionally, we must consider that TiReX execution times come from the hardware performance counters and account for the shortest matching time. These results compare fairly with those on the CPU since we ran the whole matching process inside a loop where the first ten iterations were used to warm up the L1 data and instructions caches (which are, for all CPUs, 32KB in size, hence they can host the whole input and program code), and we averaged the run-time of 30 iterations. Instead, we averaged 30 iterations and subtracted a fixed amount of time for file opening to compare fairly with Grep.

5.5.3.2 The L Scenario

In this scenario, we focus more on the throughput oriented solutions, restricting our implementation to the external host model and the AWS F1 instance, as it is the most representative for a server-like environment, ac-



(a) *Execution time (the lower, the better)* **(b)** *Energy efficiencies (the higher, the better)*

Figure 5.10: *Geometric means results and improvements of TiReX-16 VU9P with respect to other solutions on L scenario.*

counting for the overhead presented at the beginning. Similarly, we consider the Intel Xeon CPU only. We consider different datasets from the S scenario, as described at the beginning of the Section, for both REs and data to analyze.

Figure 5.10a shows the geometric means of the execution times (the lower, the better) achieved on the two benchmarks by the considered approaches, with the speedup of the 16-core version of TiReX on the VU9P reported on top of the bar. It is noteworthy that PROSITE exhibits more matching situations, while PowerEN reports few matches. TiReX single-core already offers a speedup compared to tools like Grep and FLEX, showing the benefits of TiReX domain specialization achievable at the single-core level. Scaling TiReX to a multi-core architecture delivers a speedup of 1.233$\times$ and 1.585x over state-of-the-art software such as RE2. Moreover, our DSA can reach higher performance, by further increasing the number of cores, but, as stated in Section 5.5.2, the tools and technology limit the number of AXI-Master ports we can instantiate on the VU9P.

Figure 5.10b shows the power efficiencies (the higher, the better) of the considered literature software approaches and TiReX single and multi core architectures. On top of the bars is reported the improvements of TiReX VUP9 16-corecompared to the other approaches. We consider for all the approaches the TDP of the respective devices, i.e., the Xeon (115W) and VU9P (42W [341]). The 16-core design achieves energy efficiencies that range from $\sim$3$\times$ to peak of $\sim$490$\times$.

Table 5.6: *Comparisons in terms of platform, bitrate, flexibility and benchmarks with the Related Work (in bold the best).*

Solution	Target Platform	Frequency $[MHz]$	Bitrate [*Gb/s*]	Energy Eff. $[Gb/(s \cdot W)]$	Run-time adaptability	Benchmarks
VC707 16-core	FPGA (28nm)	130	16.65 - 116.4866.61	0.78 - 53.148	✓	Synthetic, PROSITE PowerEN [290]
PYNQ 8-core	FPGA (28nm)	80	4.51 - 35.818.04	1.35 - **10.775.25**	✓	Synthetic, PROSITE PowerEN [290]
VUP9 16-core	FPGA (16nm)	202	25.94 - 180.9903.78	0.62 - 42.3147	✓	Synthetic, PROSITE PowerEN [290]
ReCPU [308]	ASIC (180nm)	318	10.19 - 18.18	N/A	✓	Synthetic
Brodie et al. [321]	ASIC (N/A)	133	16	N/A	✓	Synthetic, Snort, Forensic
BFSM [299]	PEN (45nm)	2000	20 - 40	N/A	✓	Open source REs
Meiners et al. [285]	TCAM (N/A)	N/A	10 - 19	N/A	✓	Bro, Snort, L7-filter, Networking
REAPR [159]	FPGA (20nm)	222 - 686	0.20 - 0.60	0.06 - 0.20	-	ANMLZoo [290]
FlexAmata [160]	FPGA (16nm)	163 - 213	3.70	N/A	-	ANMLZoo [290]
HARE [284]	FPGA/ASIC (28/45nm)	100 / 1000	3.20 / 256	N/A / 2.13	-	Synthetic, Snort
PiDFA [319]	FPGA (28nm)	N/A	3.39 - 29.59	N/A	-	Bro, Snort, L7-filter
jDFA [305]	FPGA (28nm)	150	230 - **430**	N/A	-	Bro, Snort and L7-filter
Atasu et al. [306]	FPGA (40nm)	106 - 250	1 - 16	N/A	-	L7-filter, Text Analytics
Agarwal et al. [332]	FPGA (40nm)	250	12.16	N/A	-	Text Analytics
Nguyen et al. [333]	FPGA (28nm)	100	11.98	N/A	-	Text Analytics

"thesis" — 2022/1/23 — 18:33 — page 107 — #119

5.5.4 Comparison to Related Work

Finally, we compare our work against the hardware-based solutions available in the literature, focusing on the top-performing ones and those exhibiting an approach close to the proposed one, as reported in Table 5.6. We consider the targeted benchmarks, the throughput in bitrate, the energy efficiency when available, and the run-time flexibility for replacing the REs. The bitrate and power numbers are taken directly from results reported by the respective authors since other implementations are not open-source, hence not replicable by us, or simulated only, while ours are derived from Section 5.3.6 best and worst case. The energy efficiency is the ratio of $throughput/power\ consumption$, but we employ the bitrate in Gb/s reported by the different approaches as throughput. Although few approaches give importance to the power consumption for the computation, we claim it is a relevant comparison metric also in this field. Indeed, being FSMs among the most relevant computational kernels in computing fields [44], and although they are intrinsically sequential, providing efficient computations is paramount [301]. Flexibility is a qualitative measurement that refers to the mutability of the matching engine based on the RE(s). We adopted a binary classification based on our understanding of the cited research work. This means that this is not an absolute and quantitative measurement but a criterion that we claim to be relevant. A non-flexible approach means that, given a different (set of) REs, the architecture needs to be re-synthesized or changed accordingly. Indeed, the time to generate a new bitstream can range from one to several hours, depending on the design complexity. Hence, it is often unacceptable if a database of ready bitstreams is not available. The benchmark column reports the kind of benchmarks used in the research work we cite. Considering the solutions that do not provide run-time adaptability for the RE(s) to look for, their obtained throughput is below or at least comparable to our high-throughput-oriented implementation. However, jDFA [305] reaches a throughput that is higher than all our solutions, but it is noticeable that it does not provide a run-time pattern change. Therefore, the time to update their system is the one required to completely regenerate the bitstream for the FPGA. It is important to notice HARE [284], which provides two implementations targeting either an FPGA or an ASIC device and proposes interesting ideas. They reach remarkable performance with the ASIC version, though it is an RTL-simulated-level design at *1 GHz*. Indeed, their scaled-down version, which is effectively deployed on a FPGA, achieves a throughput of *3.2 Gb/s*, which is lower than our PYNQ-Z1 version, not aimed at high-

throughput scenarios. Among the considered work, REAPR [159] represents an exciting tool for FPGA-based automata computations tested extensively on a broad set of benchmarks, and it is one of the few reporting the power consumption. On the other hand, other solutions provide flexibility as one of their main features. Indeed, Brodie et al. [321] ensure the run-time REs change, obtaining a throughput lower than all our solutions. The work by Meiners et al. [285] and the Power Edge of Network (or PowerEN) by IBM represent the most interesting related work exposing the software programmability on the pattern to search for and delivering good throughput, though not reporting the power consumption.

5.6 Final Remarks

This work presents a DSA, called TiReX, based on the REs as a program approach. The single-core delivers performance in line with the top software with promising energy efficiencies. We provide a multi-core architecture to increase the parallelism level at both the single RE and the multi-RE level while providing an architectural model for different workloads. We show how we reach comparable results with many state-of-the-art hardware-based solutions, providing a high degree of run-time adaptability of the RE. Our testing scenarios show we can achieve top speedup of 263× in latency, and in throughput of ∼179×. Thanks to our domain-specialization, we deliver outstanding energy efficiency results that spans from 3× to 490× than state-of-the-art software.

CHAPTER 6

A Breadth-First-based Domain-Specific Architecture for Efficient Regular Expression Matching

This Chapter describes the second Domain-Specific Architecture (DSA) for Regular Expression (RE) in this dissertation, called CICERO, which is based on an execution model closer to a breadth-first approach and an Non-deterministic Finite Automata (NFA). The previous Chapter shows promising results that, however, might suffer the intrinsic nature of a backtracking approach. Therefore, CICERO aims at overcoming the backtracking issue with a different architecture, still able to ensure flexibility and performance. Overall, CICERO comprehends an end-to-end framework composed of a DSA and a companion compilation framework for RE matching. This solution is suitable for many applications, such as genomics/proteomics and natural language processing. Moreover, CICERO aims at exploiting the intrinsic parallelism of non-deterministic representations of the REs. On top of this, CICERO can trade-off accelerators' efficiency and processors' flexibility thanks to its programmable architecture and the compilation framework.

Despite significant algorithmic improvements, software solutions cannot keep pace with the increasing size of the processed data (either input strings or Regular Expressions (REs)). For this reason, hardware acceleration is a valid alternative for computationally-intensive kernels such as those for RE matching [160, 284, 293, 298, 319, 326, 336, 342]. In this scenario, reconfigurable Field Programmable Gate Array (FPGA) devices represent a viable solution to boost the matching process while keeping a low energy profile. FPGAs can achieve a throughput of 100Gbps during intrusion prevention while a CPU with 250 cores is limited only to 400Mbps [288] (almost 250× of improvement). So, FPGAs can be used to implement specialized energy-efficient RE engines, while the device or part of it can be turned off when unused [301, 311]. However, since fixed-function accelerators embed custom RE matching logic for a given set of REs, they cannot be applied for other patterns, limiting the solution flexibility. Indeed, this approach requires to re-synthesize the logic for each new RE.

To overcome these limitations, we propose CICERO, a complete solution based on **domain-specific programmable engines** for RE matching. CICERO includes a *domain-specific architecture* for RE matching where each engine's execution model is based on Thompson's approach (Section 6.4) and a *compilation framework* (Section 6.3) to create the programming code of such engines. Indeed, given the input REs, the CICERO compiler translates it into our architecture machine code based on a simplified Instruction Set Architecture (ISA) (Section 6.2). Moreover, it applies optimizations to reduce the code size (i.e., number of instructions) and extract more hardware parallelism. Then, the CICERO engine executes such instructions while processing the input string. To exploit more hardware parallelism, we also describe a parallel architecture composed of multiple engines, evaluating two alternative interconnection topologies. CICERO combines the efficiency of specialized hardware accelerators and the flexibility of general-purpose processors. We evaluated our single- and multi-engine FPGA prototypes using real benchmarks from the open-source AutomataZoo benchmark suite [283]. We obtained excellent results both in terms of performance and energy efficiency: our CICERO architecture is 28.6× and 20.8× more energy-efficient than ARM and Intel processors, respectively (Section 6.5).

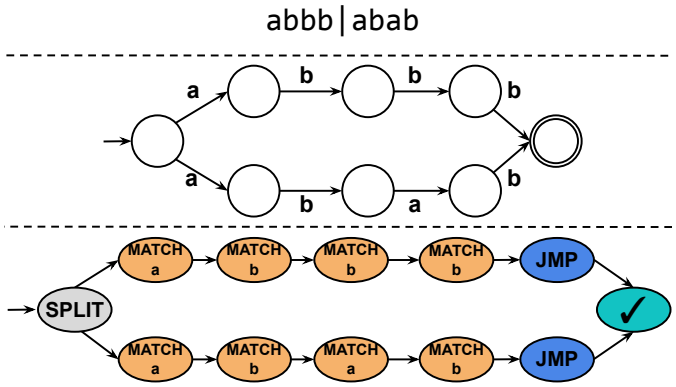


Figure 6.1: *From a RE "abbb|abab" (top) to its NFA (mid) and, finally, to CICERO instructions (bottom).*

6.1 Breadth-first Regular Expression Matching Example

This work aims at implementing an architecture for RE matching with two conflicting goals: 1) provide the efficiency of hardware accelerators thanks to specialization and parallel execution, and 2) offer the flexibility and reusability of general-purpose processors. Similarly to Section 5.1 example and approach, we address alternatives and non-determinism using parallel hardware execution flows similar to Thompson's threads. In particular, we aim at executing the threads with domain-specific engines that allow us to process the alternatives with the efficiency of hardware accelerators. To trade-off specialization and flexibility, our engines are domain-specific processors based on an Instruction Set Architecture (ISA) tailored to RE matching. Our architecture uses multiple execution flows that process the same current character in parallel with RE-specific instructions. We also provide a compiler-based framework to convert REs into such instructions.

Example: The bottom of Figure 6.1 shows the instructions flow generated to match the RE "`(abbb|abab)`". Each node represents a specific instruction that can either 1) proceed to the next one to continue the match or 2) stop the analysis when the input string is not accepted. We also have specific instructions to spawn "threads" (`Split`).

The following Sections dive into our approach, and in particular, Section 6.2 details CICERO ISA, which is the tailored software/hardware interface for creating domain-specific RE matching engines.

Table 6.1: *CICERO Instruction Set:* `PC` *is Program Counter (i.e., the memory address of the next instruction to be executed),* `cc` *is the pointer to the current character, and* `OP` *is the instruction operand.*

Instruction Class	Instruction	Operand	Description
Matching	`MatchAny`	-	`PC+1` and `cc+1`.
	`Match(OP)`	Character	Compares `OP` with `*cc`. In case of match, `PC+1` and `cc+1`.
	`NoMatch(OP)`	Character	Compares `OP` with `*cc`. In case of *no* match, *only* `PC+1`.
Control Flow	`Split(OP)`	Target Addr.	Produces two parallel execution flow: the first continues with the instruction that follows immediately after (`PC+1`), while a new one starts at the target address (`OP`).
	`JMP(OP)`	Target Addr.	Unconditional Jump to the target address `OP`.
Acceptance	`Accept`	-	Accepts if at the end of the string.
	`AcceptPartial`	-	Accepts at any point in the string.

6.2 CICERO Instruction Set Architecture

The CICERO ISA takes inspiration from the basic operations described by [49], which employs parallel *threads* working in lockstep on a sequence of characters [310], similar to a breadth-first exploration. For this reason, the CICERO engine must be capable of performing simple operations such as matching a character, creating threads, adapting the instruction flow, or ending the execution. For example, thread creation requires generating references to the instructions indicating the alternative execution paths' beginning.

Each CICERO instruction consists of 16 bits and is divided into two parts: an *opcode* (3 bits), which identifies the instruction type, and an *operand* (13 bits). The operand may have a different interpretation based on the opcode. All instructions are stored in memory and identified by an address. The execution of each instruction takes as input a character of the string and determines the subsequent instructions to continue the matching. The ISA is divided into three main classes, as shown in Table 6.1: *matching* (`MatchAny`, `Match`, and `NoMatch` instructions), *control flow* (`Split` and `JMP` instructions), and *acceptance* (`Accept` and `AcceptPartial` instructions).

Matching instructions consider the current string character. `MatchAny` instructions apply when the RE contains a wildcard (e.g., '`.`'). It consumes any character and moves to the next instruction. The `Match` instruction compares the current character with the instruction operand. If the two characters match, we move to the next instruction in the sequence. Otherwise, no further instruction is processed for this part of the flow. The

`NoMatch` instruction represents the dual of `Match` operations. Indeed, it checks if the operand and the current character do not correspond. In that case, it moves to the next instruction in the sequence without consuming the current character. Otherwise, if the characters match, it does not need to consider any further instruction, and this part of the flow is over. In this way, it is possible to check a single character multiple times (e.g., "`[^abc]`" can be represented by a sequence of three `NoMatch` instructions followed by a `MatchAny`).

Control flow instructions change the next instructions to be executed and are the basis for creating the multiple execution flows that process the alternative non-deterministic paths. A `JMP` instruction unconditionally sets a new arbitrary point to continue the execution flow. A `Split` instruction creates parallel execution flows (or threads). The first flow continues with the next instruction, while the second one starts at the address targeted by the operand.

Acceptance instructions conclude the RE matching algorithm. The `AcceptPartial` instruction affirmatively concludes the RE matching at any point of the input string, while the `Accept` instruction concludes only at the end of the string.

6.3 CICERO Compiler

Since CICERO instructions are stateless, we can not take advantage of state-of-the-art algorithms, such as register allocation, available in highly optimized compiler frameworks. Therefore, we built from scratch our own custom compiler that translates REs into executable binaries, according to Section 6.2 format. The compiler has a standard structure with three parts: front-end, mid-end, and back-end.

The front-end elaborates the input RE with an LR parser [343] and produces an abstract syntax tree. The parser does not support any back-reference operator since the expressive power required exceeds the regular languages [49]. At that point, the front-end manipulates the abstract syntax tree to produce our architecture-agnostic intermediate representation (IR).

The mid-end applies a sequence of architecture-independent IR optimizations to enhance the RE matching code, reduce the code size, and improve parallelism. Our set of optimizations includes *code restructuring* and *redundant instruction collapsing*. These optimizations mostly target sequences of `Split` instructions. The *code restructuring* reorganizes a sequence of `Split` instructions into a tree with minimal height, while *redundant instruction collapsing* merges equivalent instructions.

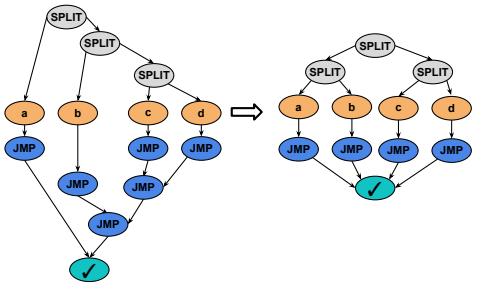


Figure 6.2: *Example of code restructuring optimization applied to the RE "(a | (b | (c | d)))".*

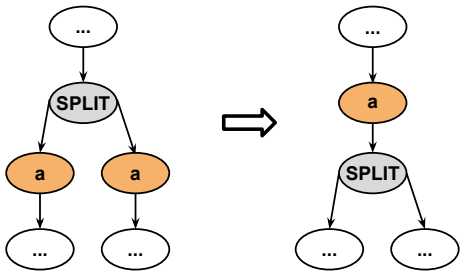


Figure 6.3: *Example of* code restructuring *optimization applied to the RE "(a...|a....)".*

Figure 6.2 shows an example of *code restructuring*. This optimization balances the number of instructions to reduce the abstract syntax tree height. Indeed, the left side of Figure 6.2 shows that the longest instruction path is three (up to the `Match 'd'`), while the path up to the `Match 'a'` contains a single instruction. Therefore, in the worst case, i.e., when the current character is 'd,' the `Match` 'd' execution happens after at least four instructions.

On the right-hand side, we can see the code after the compiler applies code restructuring. In this case, the longest path to each `Match` is equal to two. Moreover, considering a parallel architecture that can execute numerous paths simultaneously, this optimization will decrease the overall execution time. For instance, assuming each instruction is executed in a unit of time, the worst execution time with four cores is three time units.

The second optimization, i.e., *redundant instruction collapsing*, aims at identifying and merging equivalent instructions in the code. This compression reduces both code size and execution time. This is a common situation in case of non-deterministic representations, like the one in the bottom of Figure 6.1 where two equivalent operations (i.e., `Match 'a'`) follow a `Split` instruction. The compiler repeats this operation until a fixed point to compress equivalent CICERO code parts. Consider the example in Figure 6.1. We can anticipate the `Match 'a'` operations before the `Split` and collapse them into a unique equivalent instruction without modifications on the code semantics. Figure 6.3 shows an example of how this optimization reduces the size of the code, while Figure 6.5 shows the result of the optimizations to the example in Figure 6.1.

The back-end emits the actual machine instructions to be executed by the CICERO architecture. We perform code placement in memory and,

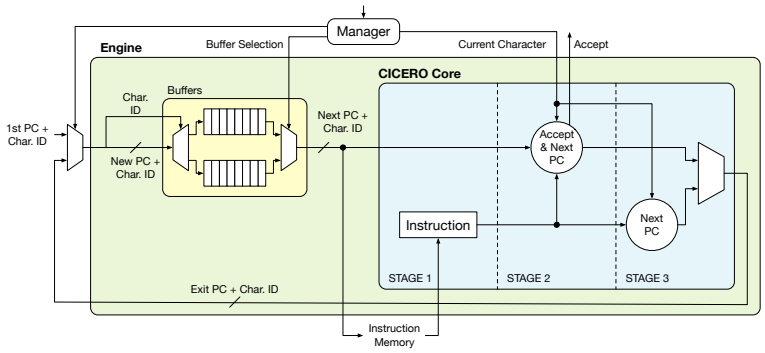


Figure 6.4: *CICERO base engine architecture.*

after that,

we apply another *redundant instruction collapsing* to `Jump` instructions. Since a chain of `Jump` instructions is inefficient (e.g., left-hand side Figure 6.2), we replace this chain with a unique `Jump`. In this way, we reduce the number of subsequent `Jump` instructions to be processed.

6.4 CICERO Architecture

This section describes the fundamental building blocks of our CICERO architecture. First, we describe the CICERO base engine that elaborates the instructions over a single character at a time (Section 6.4.1). Then, we increase the degree of parallelism in the CICERO engine enabling the ability of processing multiple characters (Section 6.4.2). Finally, we aim at further increasing the parallelism in instruction processing with a multi-engine architecture (Section 6.4.3). In this context, we also explore two different interconnection topologies that offer different scalability models.

6.4.1 CICERO Base Engine

The fundamental block of the CICERO architecture is the CICERO engine, which processes the RE instructions with a minimal amount of resources. The CICERO engine has two main components: the *CICERO Core* and the *Buffers*. As shown in Figure 6.4, we combine the CICERO engine with an *Instruction Memory* and a *Manager* module to obtain a platform that executes the instructions described in Section 6.2. The CICERO RE matching code requires executing all instructions related to a string character before moving to the next one until the string is either accepted or rejected.

The CICERO Core is a 3-stage pipelined processor that executes the in-

structions stored in the *Instruction Memory*. The *Program Counter (PC)* refers to this memory and indicates the next instruction to be executed[1]. The first pipeline stage uses the PC signal to address the *Instruction Memory* and loads the next instruction. Both the remaining stages decode and execute the instruction to either indicate the next instruction (by producing a new PC) or conclude the RE matching algorithm by raising the *Accept* signal. The CICERO Core requires these two additional stages since the *Split* instruction produces two PCs (corresponding to the beginning of the two threads), while the engine has only one output port. Clearly, the third stage is executed only for this type of instruction. The output port includes the PC and one extra bit to specify whether the thread has to continue with the current character or proceed with the next. This bit redirects the CICERO core output into the proper first-in-first-out (FIFO) of the *Buffers*. Furthermore, adding a multiplexer to the CICERO core input allows us to insert the first thread, i.e., first instruction to be executed and first character to be processed. The *Buffers* are composed of two FIFOs (or more as in Section 6.4.2). We employ them as a ping-pong buffer that contains instructions related to the current character and the other PCs for the following one.

The *Manager* selects from which FIFO the CICERO Core gets the following operation to be processed. Therefore, the *Manager* alternates the content of the FIFOs among current character PCs and following character PCs. Moreover, the *Manager* controls the overall execution of the RE matching algorithm. Once the CICERO Core has consumed all instructions related to the current character, the *Manager* provides the new character and changes the FIFO for the CICERO Core. The FIFO that is currently empty becomes the FIFO for the new next character. When the CICERO Core reaches an *Accept* instruction, the CICERO engine notifies that the string is accepted. Otherwise, when both queues are empty, the *Manager* concludes that the string does not match the RE.

Running Example Consider the RE "`abbb|abab`" in Figure 6.1 and the corresponding optimized CICERO code in Figure 6.5, together with the input string "`ababcd`". The engine initialization starts with the first thread, which has PC equal to 1, and the current character is the first '`a`'. The first instruction is a `Match` '`a`', and it is stored in the first FIFO (let us call it FIFO 0), while the other FIFO (FIFO 1) is empty. CICERO fetches the first input character, i.e., '`a`'. Then, it executes the first instruction (i.e., `Match`

[1] In the following, we use the term Program Counter (PC) and instruction, interchangeably. Indeed, the PC is the memory reference to the corresponding instruction to be executed.

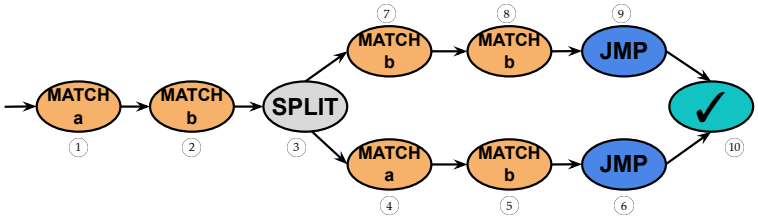


Figure 6.5: *Optimized CICERO instructions for example RE in Figure 6.1.*

	string:	a	b	a	b	c	d										
		Clock cycles															
PC	Instruction	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	...
1	match(a)	S1	S2														
2	match(b)			S1	S2												
3	split(7)					S1	S2	S3									
4	match(a)							S1	S2								
5	match(b)										S1	S2					
6	jmp(10)												S1	S2			
7	match(b)								S1	S2							
8	match(b)																
9	jmp(10)																
10	accept_partial														S1	S2	

Figure 6.6: *Execution timing diagram of CICERO code. S1, S2, and S3 indicate the stages of CICERO core.*

'a'), consumes the first 'a' of the input string, and produces the reference to the second instruction (i.e., `Match` 'b'). Since this instruction refers to the next character, the *Manager* adds it in the FIFO 1. FIFO 0, which is the FIFO of the current character, i.e., 'a', is now empty since all corresponding instructions are executed; hence, we can move to the following character of the input string, i.e., the first 'b', and switch to FIFO 1. CICERO executes the second instruction (i.e., `Match` 'b') and produces the `Split` instruction, i.e., number 3 in Figure 6.5. Given that there are no more instructions for the current character 'b', we move to the following one, i.e., the second 'a', and swap FIFO 1 for FIFO 0. The core executes the `Split` instruction and produces two instructions: instruction 4 (`Match` 'a') and instruction 7 (`Match` 'b') . Since both refer to the current character, the *Manager* adds them in the current FIFO, i.e., FIFO 0. CICERO starts executing instructions 4 and 7, but only instruction 4 (i.e., `Match` 'a') matches and produces a new instruction , i.e., number 5 `Match` 'b', which the Manager places on FIFO 1. We move ahead of one character in the string, i.e., the last 'b', and switch to FIFO 1. CICERO executes instruction 5, i.e., `Match` 'b', which produces the `JMP` instruction, number 6. As for the previous case, there are no more instructions referring to the current character, and we move forward in the input string fetching the 'c' character, and we swap the FIFOs,

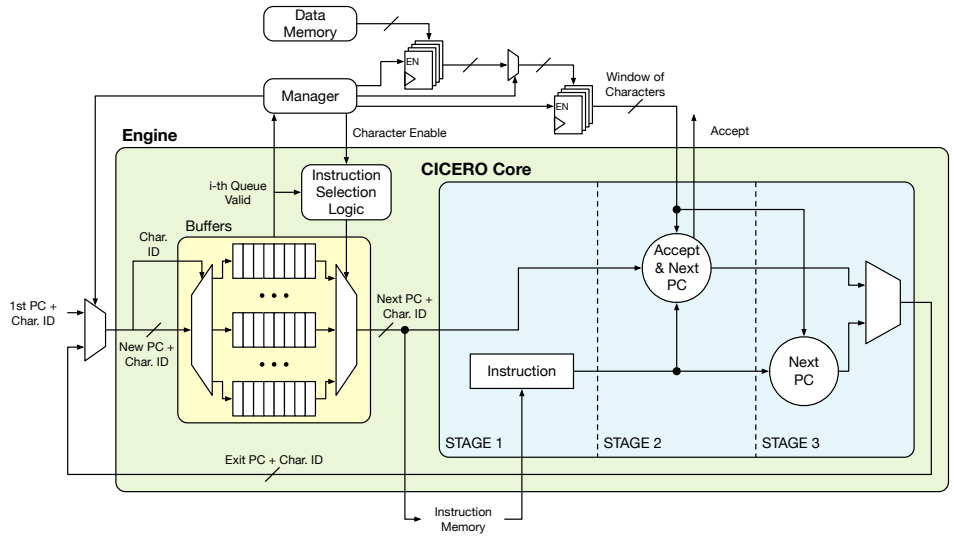


Figure 6.7: *CICERO base engine extended with multi-character support.*

i.e. FIFO 1 for FIFO 0. Since the `JMP` does not read any character, instruction 10 is pushed into the current FIFO, i.e., FIFO 0. Finally, CICERO takes instruction 10 from the queue and executes the `AcceptPartial`, ending the overall matching procedure. Figure 6.6 shows the execution timing diagram of the described running example.

As the reader can see from this diagram, there are no instructions with different colors (i.e., referring to different characters) executed in the same clock cycle, even though they may be ready to execute. For instance, instruction 5 is ready to be executed at clock cycle 8, however, its execution is postponed at the end of all instruction related to third character. This execution delay will play a crucial role in the following section of the manuscript.

In standard processors, supporting threads requires that the thread context is saved when it moves to the idle state and reloaded once the thread is resumed. In CICERO, the threads refer to the parallel flows processing the current character. Since the CICERO Core does not produce any temporary values nor stores value in a register file, the CICERO *context* includes only the PC and the current string pointer. The current character is shared among all active threads; hence, the PC provides enough information to restart the corresponding thread.

6.4.2 CICERO Multi-Character Engine

The engine described in the previous section has an architecture able to process a single character with multiple threads working in lockstep on a sequence of instructions (i.e., it consumes a character for each possible instruction flow). In this way, CICERO works in a breadth-first style that consumes a character at a time without backtracking, similar to a single-stride NFA (i.e., single character [319]), with two buffers. However, though it has noteworthy abilities, the single character consumption rate limits the achievable throughput of characters processed per second.

Though we adopt an algorithmic approach that is more efficient than backtracking, the system considers all the current character's execution paths before moving to the next. The effectiveness of this approach is high whenever compared to backtracking or processing workloads containing several parallel sub-expression to evaluate. As the engine utilization increases, we extract the best from CICERO. However, this is not always the kind of workloads in the RE world [290], and sequential execution is inevitable [44]. There are two possible approaches: dealing with thread accumulation in the next character buffer or increasing the character processing rate.

Considering the second approach, we can enhance the architecture by analyzing a character window of 2^W characters, i.e., 2^W-stride NFA, with parallel threads in lockstep, as shown in Figure 6.7. In this way, we can keep code portability among different windows of engines (i.e., the modification is not visible at the ISA level), but we increase the engine character consumption rate that can now run on $2^W - 1$ parallel characters. However, the thread context has to be updated to keep track of the consumption pointer of the input string. CICERO handles this optimization by employing a W-bit ID, called `CC_ID`, that refers to the current character in the window analyzed. Moreover, whenever we encounter `Match` or a `MatchAny` instructions, we should update `CC_ID` to reflect the fact that we moved to the next character. Considering that the `CC_ID` is a natural number modulo 2^W, CICERO keeps the threads of the last character of the window in a non-ready state. Indeed, their execution might conflict with a thread with the same `CC_ID` that refers to a newer character.

For instance, consider the input string "abcde", $W = 2$, and a current window "abcd" within CICERO, with running threads with `CC_ID`'s 0, 1, 2, 3. If the thread with `CC_ID` 3 goes first and finds an instruction that consumes a character (e.g., `Match` or `MatchAny`), the thread `CC_ID` increases and become 0, i.e., (3 + 1) mod 2^2. At this point, this thread

would wrongly target the 'a' in the buffer instead of the proper 'e'. For this reason, we stall threads related to the last character of the window that explains the minus one in the $2^W - 1$ parallel characters.

To summarize, the proposed optimization is a *sliding window* that the *Manager* handles as a circular buffer. Indeed, we need to add a FIFO for each character of the sliding window (i.e., if $W = 3$ we need a total of 8 FIFOs) to handle separate thread contexts and adopt a policy that executes always the oldest thread. An arbiter with programmable priority will let the proper instruction execute. Therefore, we can consider the CICERO base engine described in the previous section as a particular case with a window $W = 1$.

Finally, we need to account for the logic required to slide the window ahead, i.e., moving ahead of one character. Indeed, the enhanced architecture tracks the number of threads per `CC_ID` in flight in every architecture component (e.g., *Buffers*, CICERO Core, engine). Practically, there is a 2^W-bit wide bitmap that has an i^{th} active bit if there exists at least an active thread with `CC_ID` $= i^{th}$ in the architecture. The bitmaps are then combined with bitwise OR operations to hint the *Manager* on sliding the window or not. Indeed, if the character bit closest to the beginning of the window, i.e., the oldest one, is unset, the *Manager* fetches another character and slides the window.

Running Example To better illustrate the mechanism behind CICERO with Multi-Character Engine and how it takes advantage of non-determinism, we consider an extension to the example of Section 6.4.1. Consider the RE ".*(abab|abbb)", which is shown in Figure 6.8 in the form of a CICERO code, and consider as input string "abaababd". We chose this RE because it also highlights how CICERO manages non-determinism conversely to a backtracking approach. Indeed, CICERO adopts a breadth-first like execution model that explores all the alternatives at the same level. The inherent non-determinism in the considered RE leads CICERO to execute instructions 1, 2, 3, and 4 for every character in the string to test the actual starting point of the matching procedure, which begins at instruction 4. Moreover, since CICERO instructions do not rely on state, it can start considering new instructions that are ready to execute while running instructions 1,2,3 and 4. The reader can appreciate a graphical representation of this effect in Figure 6.9, which compares the pipeline of Single-Character CICERO (i.e., $W = 1$), on the top, with a Multi-Character (i.e., $W = 2$) CICERO, at the bottom. This part of Figure 6.9 shows that by supporting up to four (i.e., 2^W, where $W = 2$) parallel characters, CICERO avoids waiting for the

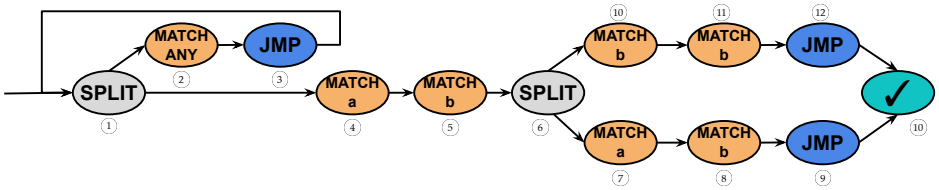


Figure 6.8: *CICERO code corresponding to ".*(abab|abbb)".*

string: a b a a b a b d

	W=1	Clock cycles																		
PC	Instruction	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17	...
1	split(4)	S1	S2	S3					S1	S2	S3									
2	matchany			S1	S2						S1	S2								
3	jmp(1)						S1	S2							S1	S2				
4	match(a)				S1	S2						S1	S2							
5	match(b)							S1	S2											
6	split(10)													S1	S2	S3				
7	match(a)																S1	S2		
8	match(b)																			
9	jmp(13)																			
10	match(b)																			
11	match(b)																			
12	jmp(13)																			
13	accept_partial																			

string: a b a a b a b d

	W=2	Clock cycles																		
PC	Instruction	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17	...
1	split(4)	S1	S2	S3				S1	S2	S3					S1	S2	S3			
2	matchany			S1	S2					S1	S2						S1	S2		
3	jmp(1)					S1	S2						S1	S2						
4	match(a)				S1	S2					S1	S2						S1	S2	
5	match(b)						S1	S2												
6	split(10)								S1	S2	S3									
7	match(a)											S1	S2							
8	match(b)															S1	S2			
9	jmp(13)																			
10	match(b)													S1	S2					
11	match(b)																			
12	jmp(13)																			
13	accept_partial																			

Figure 6.9: *Comparison of CICERO execution timing diagrams with Single character Engine (W=1) and with Multi-Character Engine (W=2). S1, S2, and S3 indicate the stages of CICERO core.*

pipeline flush before processing the new character in the input string. For instance, instruction 6 can execute at clock cycle 7 (bottom of the Figure),

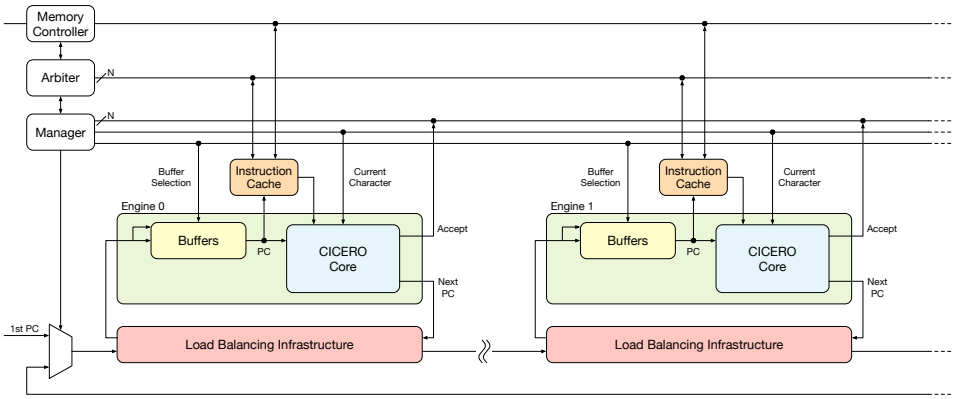


Figure 6.10: *An overview of CICERO multi-engine architecture showing the overall infrastructure that wraps CICERO engines.*

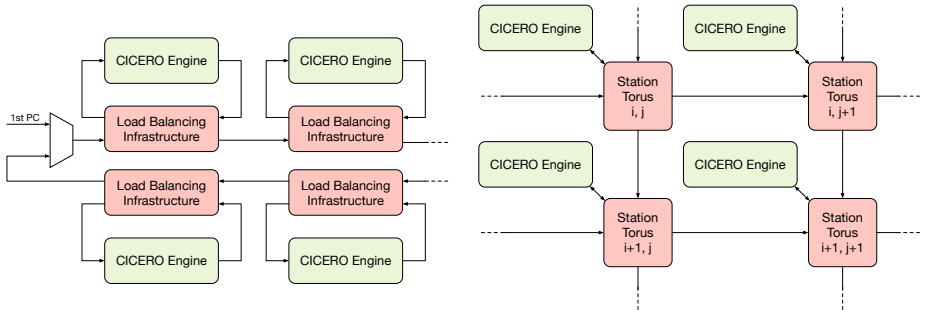
instead of waiting for the end of the processing of character 'b' at clock cycle 12 (top of the Figure). Thanks to this improvement, we can increase the pipeline occupancy (in the example, we move from an instruction per clock of 11/17=64% to 16/18=88%). Consequently, the proposed optimization reduces execution times and increases the character processing rate.

6.4.3 CICERO Multi-Engine Architecture

In the previous sections, we described the base design of the CICERO engine together with the sliding window implementation. However, with our execution model we can exploit a further degree of parallelism related to the instructions of the threads. Since CICERO instructions do not have side effects, they can be safely executed in parallel by multiple CICERO engines to increase the parallelism. The parallel version of CICERO features multiple CICERO engines with a centralized *Manager* and a distributed *Load Balancing Infrastructure* as shown in Figure 6.10.

As discussed above, the *Manager* supplies the current character to the engines and makes decisions on the overall matching process. It decides when to move to the next string character (or slide the window), accept a RE if one of the engines notifies an accept, and reject the RE after consuming all the instructions. To support parallel execution, we add a private block-based instruction cache to each engine. Since the instruction memory is read-only, no coherency protocol is needed. If a cache miss occurs, a round-robin *Arbiter* regulates the access to the *Instruction Memory*.

The *Load Balancing Infrastructure* handles the thread's execution on different engines without affecting the critical path. Each CICERO engine



(a) *Ring topology.* **(b)** *Torus topology.*

Figure 6.11: *CICERO interconnection topologies.*

features an instance of the *Load Balancing Infrastructure*. It consists of a *Station* and a latency insensitive *channel* [344]. At each clock cycle, the Station receives as input the engine output (i.e., the thread information composed of PC and a `CC_ID`), the number of instructions inside the local engine, and information from the nearby stations. At the same time, the *Station* obtains the expected latency of running an instruction that might flows to the next *Load Balancing* instances. Moreover, the *Station* can receive threads to execute from the previous stations. Based on the number of buffered instructions in the local engine (i.e., the local latency), and the latency coming from the next Station, each *Load Balancing Infrastructure* decides where to move the CICERO output and the threads by computing the minimum latency among the possible paths. Finally, the *Load Balancing Infrastructure* computes its input latency as the minimum between the number of threads to execute in the CICERO engine and the latency coming from the next Station. The latency information is then adjusted to consider the number of threads in flight along the channel. To avoid any combinational path, registers separate the latency on every channel.

We devise two different topologies for the multi-engine architecture. The first topology is a ring where each engine connects with the other two engines at most, as in Figure 6.11a. The second topology is a torus where each engine connects with at most four other engines, as in Figure 6.11b. While the ring is a simple topology but with limited scalability, the torus ideally provides a more scalable interconnection topology since each engine has more alternative where to send the threads. In both cases, we need a multiplexer to initialize the multi-engine architecture with the first thread.

Considering the **ring topology** (Figure 6.11a), the *Station* modules and the *Load Balancing Infrastructure* ones are equivalent to the ones described

in the previous section. *Station* modules are connected through latency-insensitive queues to form a ring. This protocol guarantees correct execution in all cases. In this way, we aim at evenly distributing the number of threads to elaborate across the engines.

For the **torus topology** (Figure 6.11b), we can reuse the ring topology's interconnection components to design an XY-distributed *Load Balancing* station (called *Torus Station*) on top of the *Ring Station*. The X_{in} input flows into a ring-based *Station*, and the link with CICERO engines remains the same. The first ring-based *Station*'s output then passes to the second ring-based *Station* via a latency-insensitive queue. In the second *Station*, we have the additional input, Y_{in}, which produces two outputs, namely X_{out} and Y_{out}. In this way, we can link the *Ring Stations* as in Figure 6.11b.

6.5 Experimental Validation

We implemented CICERO in SystemVerilog with a standard AXI interface and created FPGA prototypes exploiting the Xilinx Vivado HLx 2019.2 toolchain. We targeted an embedded FPGA board, namely the Xilinx Ultra96v2 (Zynq Ultrascale+ MPSoC XCZU3EG A484), on which we employ the PYNQ framework [61].

At first, we analyzed the impact of compiler and architectural optimizations on CICERO performance (Section 6.5.1). In particular, we measured the code size and clock cycle reductions that the implemented compiler optimizations enabled. Similarly, we investigated the matching time and energy efficiency benefits that the multi-character and multi-engine approaches provide when targeting FPGA with running frequency of 200MHz. Finally, we compared our best FPGA prototype against Google RE2 [334], an optimized multi-threaded C++ library for RE, in terms of matching time and energy efficiency (see Section 6.5.2).

In all the experiments, we used Protomata [345] and Brill [291] benchmarks from the AutomataZoo suite [283], which represent proteomics and natural language processing applications, respectively. We considered Protomata and Brill since they both belong to the family of "Regex" benchmarks of the original ANMLZoo suite [290]. Therefore, their RE representation is ready to use [283], and they target novel compelling research fields, i.e., bioinformatics and natural language processing. Moreover, we believe that these two benchmarks represent two opposite use cases: one more suitable to CICERO features, i.e., with a high number of alternatives (Protomata), against an unsuitable one with a wide variety of sequential REs, (Brill). AutomataZoo reports an active set (i.e., the average number

of active states in the NFA) of 712 for Protomata against 78 for Brill [283]. Indeed, most Protomata REs include many non-contiguous character sets to test. In this way, a generic architecture has to evaluate a more significant number of alternative paths/sub-expressions, and partial matches (part of the string matches the initial part of the RE) are more likely to happen. If adopting a backtracking approach, the target platform will most likely suffer from it and obtain poor performance on Protomata. Instead, Brill contains a mix of contiguous character sets and sequential matches. This second benchmark brings more advantages to traditional von Neumann architectures, which can handle these sets with simple arithmetic differences, sequential executions, and aggressive approaches similar to backtracking. We exclude other benchmarks since they either provide the automaton instead of the RE or contain unsupported features of non-regular languages such as backreferences. We evaluated these benchmarks on the same suite's input and applied the RE matching to at most 1,024 characters if the input string was bigger. We also combined some REs in the two benchmarks to increase the RE complexity. To do so, we used up to four operators '|' to create parallel alternatives. These combined REs increase the number of alternative paths simultaneously active and provide a scenario where the final user aims to match a set of REs in a single input pass. We randomly sampled 1,000 REs from both Protomata and Brill, and combined four different random REs together in a combinatorial manner, i.e., providing all the possible combinations, as previously described. Then, we randomly sampled 200 combined REs from this new set of REs and 200 possible input strings from the original AutomataZoo. We will call these combined versions *Protomata4* and *Brill4* benchmarks.

Throughout the evaluation, we employ three different sets of tests for the considered benchmarks. The first one is a subset of REs and inputs randomly sampled with a uniform distribution from the original benchmarks. The second subset contains 200 REs randomly sampled with a uniform distribution from the combined benchmarks (i.e., Protomata4 and Brill4) to increase the parallelism degree and better highlight the benefits of a multi-engine CICERO. The third set comprises the complete benchmark tests as published by the suite authors to provide a fair comparison with other approaches employing established test suites.

6.5.1 Evaluation of Compiler and Architectural Optimizations

This Section describes the impacts of compiler and architectural optimizations on CICERO performance. First of all, we analyze the benefits that

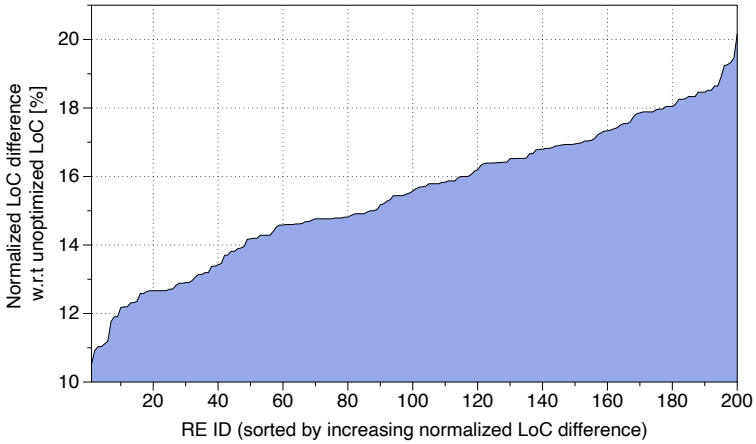


Figure 6.12: *Unoptimized vs optimized Lines of Code (LoC) difference normalized w.r.t. the unoptimized LoC size ($\frac{Unopt_{LoC}-Opt_{LoC}}{Unopt_{LoC}}$) on REs sampled from Protomata4, showing the improvements w.r.t. the original LoC.*

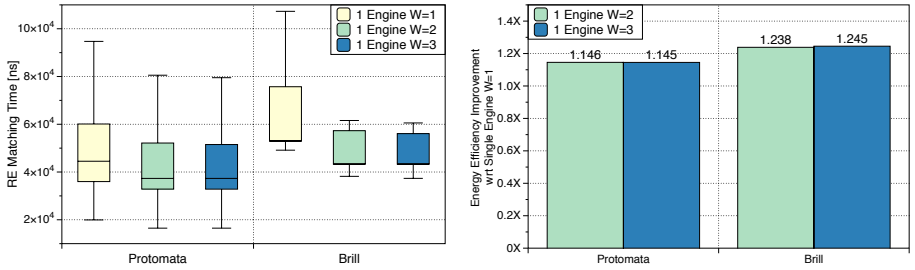
the compiler optimizations introduce in terms of code size and processing time (Section 6.5.1.1). Then, we evaluate how CICERO performance scales according to the character window (Section 6.5.1.2) and the number of engines (Section 6.5.1.3). Finally, we examine which is the most suitable interconnection topology for the CICERO multi-engine architecture (Section 6.5.1.4).

6.5.1.1 Compiler Optimizations

Figure 6.12 shows the reduction in terms of Lines of Code (LoC), or instructions, among the code sizes of the Protomata's REs compiled with and without the optimizations. We normalized the difference between unoptimized and optimized LoC by the unoptimized size ($\frac{Unopt_{LoC}-Opt_{LoC}}{Unopt_{LoC}}$). On average, the optimizations save 15.48% instructions for the Protomata4 combined REs, while Brill4 has an average reduction of 1 instruction, and hence it is not plotted. Protomata code size reduction leads to a geometric mean (geomean) speedup of 1.3× compared to the unoptimized code.

6.5.1.2 Character Window Scaling

Moving to the architectural enhancements, we start evaluating the impact of the increment in character processing rate, i.e., the character window (Section 6.4.2), against the base engine (Section 6.4.1). For this analysis, we employ the standard Protomata and Brill benchmarks, and randomly



(a) *Weighted average distributions of matching times.* **(b)** *Geomean energy efficiency improvements.*

Figure 6.13: *The effects on queue scaling on a CICERO single engine (on 200 inputs and 200 REs sampled from Protomata and Brill).*

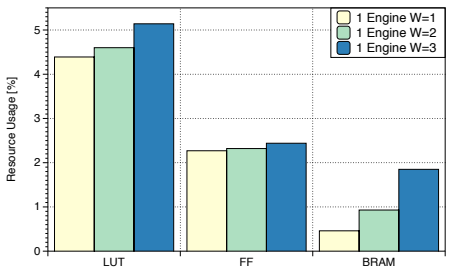
sampled 200 REs and 200 possible input strings to showcase the behavior on random REs. We measured the matching times of the FPGA prototypes through CICERO performance counters after loading the code and the string to match on CICERO memory. Besides, we extracted the board-level power consumption with the Voltcraft Energy-Logger 4000, which measures the board voltage, current, and power directly from the plug, and then computed the power consumption geometric means. Figure 6.13a shows the boxplot distributions of weighted average matching times of a CICERO single engine with windows of 2^W characters, where W is equal to 1, 2, 3. We chose the weighted average because it assesses the RE processing times better than raw runtimes as it also accounts for the processed characters. For each RE, we compute the weighted average as follows: $\frac{\sum_{i=0}^{N} MatchingTime_i \cdot l_i}{\sum_{i=0}^{N} l_i}$, where N is the number of input strings, and l_i the string length. In particular, according to the RE matching process outcome, we weigh the matching times with different input string lengths. If the RE does not match, we pick the whole string length; otherwise, as a match may occur at an arbitrary point of the input string, we approximate the number of processed characters with half of the string length.

The reader can notice from Figure 6.13a that moving from $W = 1$ to $W = 2$ reduces both the median (the black line within the colored box) and the height of the box, decreasing the average case and achieving steadier matching times. Comparing $W = 2$ and $W = 3$, the boxplot of the greater window (the blue one, $W = 3$) has a smaller length than the green one ($W = 2$). Therefore, it provides a steadier weighted matching time. However, this chart displays minimal differences among single CICERO engines with $W = 2$ and $W = 3$; therefore, we will still consider both

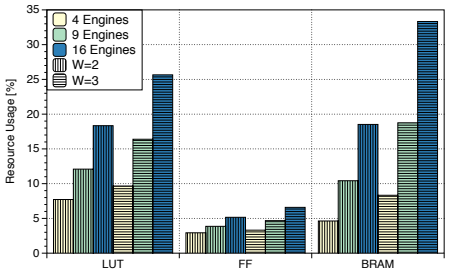
configurations. Then, we computed the geomean of energy efficiency improvements when increasing the character window against the base engine with $W = 1$. To do so, we use the previously employed weighted matching time per RE, compute the energy efficiency $\frac{1}{(WeightdMatchTme[ms] \times Power[W])}$, and finally the geomean. Figure 6.13b highlights that the CICERO single engine with $W = 2$ (i.e., the green one) is slightly more efficient than the one with $W = 3$ on Protomata, while it is slightly worse on Brill. Both Figure 6.13a and Figure 6.13b show that the increase in the window dimension gains practical improvements in median matching time, achieving steadier matching times and better energy efficiency. Figure 6.14a shows the resource utilization of the entire Ultra96v2 board, including both the CICERO engine and the additional logic that connects the engine to the ARM processor. We can notice that the CICERO engine is mainly LUT and BRAM demanding. Indeed, their usage grows according to W, as the engine needs further logic to manage the additional number of alternative paths. On the other hand, FF growth is more restrained since CICERO mainly employs FFs for the Manager state machine. Finally, the chart indicates that a CICERO engine has a low resource usage; indeed, even when W=3, the engine requires at most 5% of LUTs.

6.5.1.3 Engine Scaling

We analyzed the scaling effectiveness of CICERO multi-engine architecture, by employing randomly sampled combined benchmarks (i.e., Protomata4 and Brill4) to increase the cores utilization. In this way, we aim at showcasing the impact of scaling to multiple engines with a ring topology. Figure 6.15a shows the geomean of the speedups achieved by 4, 9, and 16 cores with $W = 2$ (vertical lines) and $W = 3$ (horizontal lines) against the CICERO single engine with $W = 2$. For Both Protomata4 and Brill4, we obtain a speedup that scales with the core number. Conversely, Figure 6.15b displays the geomean of energy efficiency improvements at the core scaling on the same benchmarks and shows how the efficiency improvements do not reflect the speedups. Indeed, considering Protomata4, the most energy-efficient architecture has four engines with both W = 2 and W = 3. However, Brill4 indicates that the most energy-efficient architecture has nine engines with W = 3. These results prove how different architectures provide different trade-offs from both matching time and energy efficiency perspectives, depending not only on the kind of REs, but also on the input string. For these reasons, from now on, we will consider only the architecture with nine engines and $W = 3$ as the reference one,

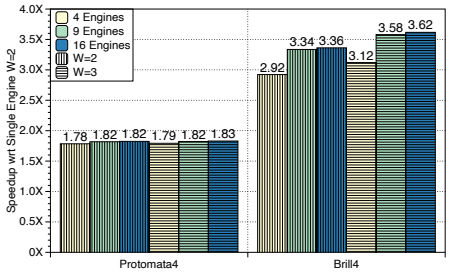


(a) *Single engine, with different window sizes.*

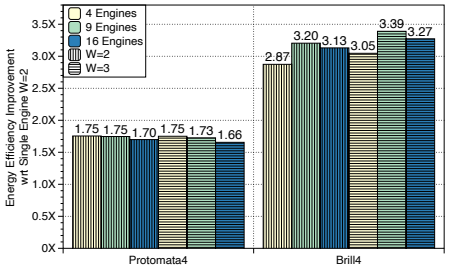


(b) *When scaling the number of engines (from 4 to 16) considering W=2 (vertical lines) and W=3 (horizontal lines).*

Figure 6.14: *Resource usage* $[\%]$



(a) *Multi-engine geomean speedups.*

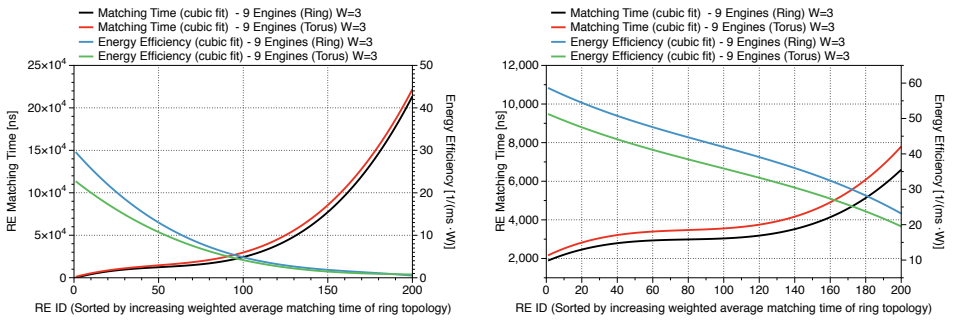


(b) *Multi-engine geomean efficiency improvements.*

Figure 6.15: *The effects on CICERO engine scaling (on 200 inputs and 200 REs from Protomata4 and Brill4).*

being the optimal trade-off among matching time and energy efficiency.

Figure 6.14b reports how the resource usage scales according to the number of engines and W. While the FF utilization remains relatively low (almost 8% in the worst case), the number of LUTs and BRAMs significantly increases due to the additional logic and memory required by both the engines and the load balancing infrastructure. This behavior is particularly evident when considering the sixteen-engine configuration. However, since such a configuration does not provide relevant performance benefits compared to a nine-engine one, there is no point in selecting it. On the other hand, even though $W = 3$ requires more resources than $W = 2$, the higher speedup and energy efficiency (especially on Brill4) compensates for the additional resources. This analysis further supports our choice of the nine-engine architecture with $W = 3$ as the reference one.



(a) *Matching times and energy efficiencies on Protomata4 benchmark.*

(b) *Matching times and energy efficiencies on Brill4 benchmark.*

Figure 6.16: *Weighted avg. matching time and energy efficiency of the topologies (on 200 inputs and 200 REs from Protomata4 and Brill4).*

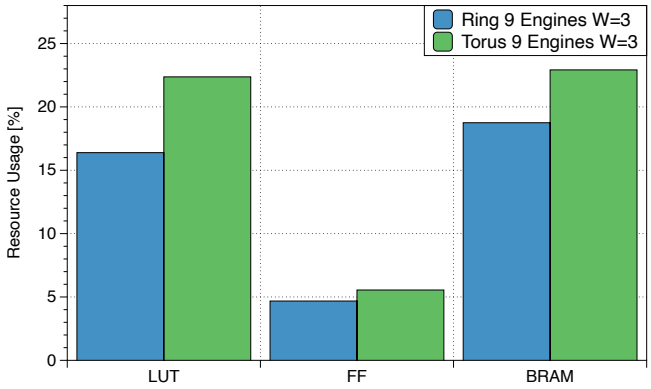


Figure 6.17: *Resource usage when considering a 9-Engine architecture with W=3 but with different topologies.*

6.5.1.4 Topology Analysis

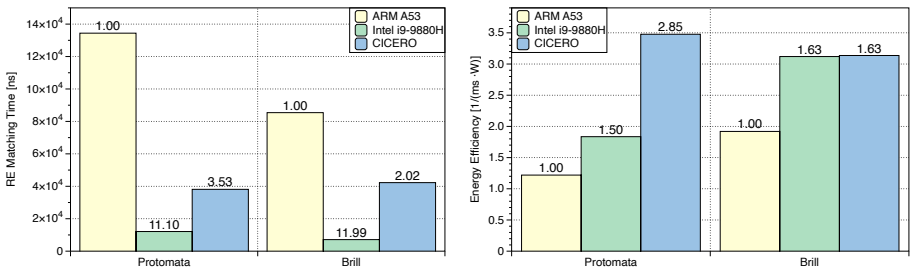
Before diving into the comparison with other literature approaches, we compare the ring topology against the torus one for our reference architecture of nine engines and $W = 3$. We exploited the same benchmarks as before, i.e., Protomata4 and Brill4, and compare the matching times and energy efficiencies of both topologies. Figure 6.16a and Figure 6.16b show the cubic fits of these measures. In this way, the reader can see that, on Protomata4 benchmark, the torus curve (the red line) always remains above the ring one (the black line) as the RE matching time increases. Considering the efficiency curves, even though the ring (the blue line) generally shows a better energy efficiency, the torus (the green line) performs slightly

better over the most time-demanding subset of REs, though it is a very restricted subset. Moving to the Brill4 results in Figure 6.16b, the cubic fits of matching times and energy efficiencies demonstrate that the ring gains better matching time (i.e., the black line stays below the red one) and achieves a higher efficiency (i.e., the blue line stays above the green one) for all the considered REs. In conclusion, although the torus represents a better solution in terms of scalability and a good candidate for a more suitable layout design, these results prove that the simplicity of ring topology is enough to keep up with both the matching time and energy efficiency of Protomata4 and Brill4 benchmarks. Besides, as shown in Figure 6.17, the ring topology is also less resource-demanding than the torus one.

6.5.2 Comparison Against Google RE2

As mentioned before, our analysis identified the ring-based nine-engine architecture with $W = 3$ as the most efficient one. This implementation requires 11,563 (16.39%) LUTs, 6,600 (4.68%) FFs, and 81 (18.75%) BRAMs on the Ultra96v2 FPGA. We compared our implementation with Google RE2 executed on two candidate processors: an embedded solution, the ARM Cortex A53 (mounted on the Ultra96v2), and a mainstream one, the Intel i9-9880H. The RE2 library was built from sources [334] with `-O3` optimizations. We set the comparison on partial match operation in cold-start conditions and measured matching time and energy efficiency for the matching process only. As previously stated, we relied on the CICERO performance counters to measure the matching time after loading the code and the string to match on CICERO memory. We used the C++ chrono library to measure the execution time of a RE2 code snippet that interprets the RE by creating an `RE2::RE2` object and calling the `RE2::PartialMatch` function. We repeated the entire procedure 30 times for both Intel and ARM CPUs to mitigate cache effects and acquire statistically relevant results. As before, we measured and weighted the average execution times on the string length analyzed. Finally, we selected the Thermal Design Power (TDP) of the Intel CPUs as reference power consumption. For the ARM CPUs and the CICERO FPGA prototype, we employed the Voltcraft Energy-Logger 4000 to extract the board-level power consumption.

Figure 6.18 shows the geomean values of the matching times (left-hand side) and energy efficiency results (right-hand side) achieved by the A53, i9, and CICERO on all the possible RE-input couples from Protomata and Brill benchmarks. The results in Figure 6.18a show that CICERO achieves lower matching times than the embedded processor (i.e., the A53) with



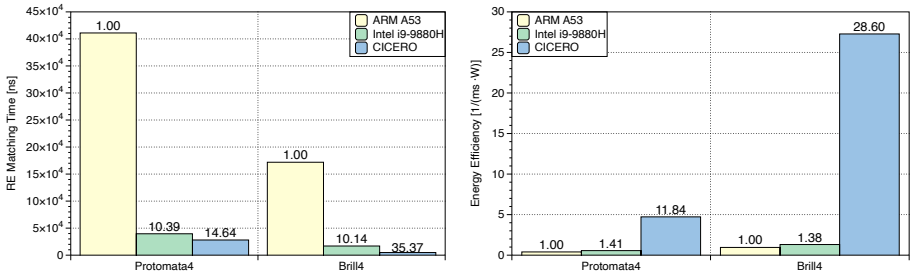
(a) *Matching times (the lower the better) and speedups wrt ARM A53.*

(b) *Energy efficiencies (the higher the better) and improvements wrt ARM A53.*

Figure 6.18: *Matching times and energy efficiency geomean of ARM A53, Intel i9, and CICERO on the complete Protomata and Brill.*

speedups of $3.526\times$ and $2.021\times$ on the Protomata and Brill, respectively. However, the i9 shows matching times even better than ours in both benchmarks. Instead, considering the energy efficiencies achieved, Figure 6.18b highlights comparable results of CICERO and the mainstream processor (i.e., the i9) on the Brill benchmark, i.e., values are around 3.136 and 3.119 $[\frac{1}{ms \cdot W}]$, respectively. However, Protomata benchmark shows the CICERO advantage of tailoring the architecture for a higher energy-efficient computation. Indeed, CICERO delivers $1.89\times$ and $2.851\times$ energy efficiency improvements than the i9 and the A53, respectively.

Overall, these results exhibit remarkable matching times and higher efficiency with the standard benchmarks as they are. However, our approach, built on Thompson's algorithm, which can scale linearly in the string length without paying the cost of alternative paths backtracking. For this reason, we collect the results of a combined version of the benchmarks as previously described. This combination increases the alternative paths simultaneously active and better mimics a real scenario where the final user aims to search all the REs in a single input pass. Indeed, considering the analysis of gigabytes of data, an optimized search wants to reduce as much as possible the number of times to scan the input data. Figure 6.19 presents the results of these combined experiments on the considered architectures. While the i9 matching times hold the same magnitude order as the standard benchmark and the A53 shows deterioration, CICERO reveals dramatic improvements as in Figure 6.19a. Indeed, the speedups achieved by the i9 and CICERO over the A53 are $10.386\times$ and $14.642\times$ on Protomata and $10.144\times$ and $35.370\times$ on Brill. While comparing the i9 with CICERO, our architecture delivers speedups of $6.173\times$ and $3.487\times$ against



(a) *Matching times (the lower the better) and speedups wrt ARM A53.*

(b) *Energy efficiencies (the higher the better) and improvements wrt ARM A53.*

Figure 6.19: *Matching times and energy efficiency geomean of ARM A53, Intel i9, and CICERO on 200 inputs and 200 REs from Protomata4 and Brill4.*

the mainstream architecture on Protomata and Brill, respectively. Moving to Figure 6.19b and the associated energy efficiency results, CICERO FPGA-based implementation presents outstanding results. Especially, CICERO achieves $8.409\times$ and $20.798\times$ energy efficiency improvements than the most energy-efficient CPU (i.e., the i9) and $11.839\times$ and $28.600\times$ than the A53 over Protomata and Brill, respectively. The domain specialization of the architecture combined with Thompson's approach leads to remarkable speedups and energy-efficient computations against both an embedded and a mainstream processor.

6.6 Related Work

As demonstrated in the previous chapter a quantitative comparison is unfair, given the extremely differences in the employed approaches. Thus, we provide a qualitative analysis, as reported in Table 6.2.

As from Section 5.2, Stream-Dataflow Architecturess (SDAs) usually rely on spatial reconfigurable architectures and efficiently compression of the Deterministic Finite Automata (DFA) delivering remarkable performance at the high cost of resynthesis the hardware for new patterns [159, 160, 284, 304, 305, 319, 326]. CICERO is, instead, a specialized but flexible architecture that can support several different REs thanks to the custom ISA and the compiler-based framework. Therefore, our approach enables generating a new binary code with software compilation time instead of waiting hours for generating a bitstream. Thus, comparing these two diverse approaches would not result in a fair apple-to-apple comparison.

Generally, In-Memory Architectures (IMA) or Software-Programmable

Table 6.2: *Related Work Summary*

Work	Target Scenario	Device	Architecture Type	Execution Mode	Compilation Framework	Compilation Time Required
CICERO-FPGA	Embedded	XCZU3EG (16nm)	Software-Programmable	NFA	Yes	Software like
FPGA [319]	Datacenter	XC7VX690T (28nm)	Stream-Dataflow	DFA	No	Bistream like
FPGA [305]	Datacenter	XC7VX1140T (28nm)	Stream-Dataflow	DFA	No	Bistream like
FPGA/ASIC [284]	Embedded	Arria V SoC (28nm)/ ASIC (45nm)	Stream-Dataflow	DFA	No	Bistream like/N.A.
FPGA [159]	Datacenter	ADM-KU3 (20nm)	Stream-Dataflow	NFA	Yes	Bistream like
FPGA [304]	Datacenter	XCVU9P (16nm)	Stream-Dataflow	NFA	Yes	Bistream like
FPGA [160, 326]	Datacenter	XCVU9P (16nm)	Stream-Dataflow	NFA	Yes	Bistream like
TCAM [285]	Datacenter	N.A.	Software-Programmable	DFA	N.A.	Software like
PowerEN [299, 336]	Datacenter	ASIC (45nm)	Software-Programmable	DFA	Yes	Software like
FPGA [48]	Datacenter	XC7VX485T (28nm)	Software-Programmable	DFA	Yes	Software like
AP [160, 290, 304, 327]	Datacenter	DRAM (22nm)	Software-Programmable	NFA	Yes	Software like

Architecturess (SPAs) are platforms more similar to CICERO. Indeed, they can update the set of REs without changing the underlining architecture, nor the bitstream [48, 285, 299, 336]. All these methods are based on the DFA representation, which grows exponentially with the complexity of the RE. Instead, we focus on an NFA implementation to allow parallelization of the alternatives. The algorithmic approaches are fundamentally different, though semantically equivalent, and the NFA implementation proved to lead execution times linear in the string length.

The Automata Processor (AP) was an outstanding spatial reconfigurable architecture that embedd an automaton into the reconfigurable fabric [327, 328], although it was a promising solution [159, 160, 291, 316], only simulation results were reported [304], in contrast with our FPGA prototype.

6.7 Final Remarks

This Chapter presented CICERO, a software-programmable, Domain-Specific Architecture (DSA) for REs matching. CICERO exploits Thompson's algorithm to create a non-deterministic RE representation that can execute on multiple engines in parallel without backtracking. We also provide an end-to-end framework for translating REs into optimized code. We validate CICERO architectural optimizations on an embedded FPGA on benchmarks from AutomataZoo [283], showing increasing benefits in the proposed solution, e.g., from the code size to the matching times and energy efficiencies. CICERO multi-engine and multi-character architecture shows up to 28.6× and 20.80× energy efficiency improvements against the highly optimized Google RE2 library onto an embedded processor (the ARM A53) and a mainstream processor (the Intel i9), respectively.

CHAPTER 7

Conclusions and Future Research Directions

This Chapter draws the conclusions, sums up the most important takeaways of this dissertation, highlights the limitations of the presented approaches, and paves the way to open new research paths.

The *domain-specialization* direction showcases an up-and-coming alternative to ever-increasing energy and performance requirements. Indeed, many commercial Domain-Specific Architectures (DSAs) are already on the market (e.g., Apple M1 Neural Engine [346], Pixel Visual Core [12], Tensor Processing Unit (TPU) [11], MSR Neural Processing Unit (NPU) [347]), and academics are pushing this research constantly [9, 17, 52, 231, 239, 241, 348, 349]. Within this context, this dissertation focuses on the role of Reconfigurable Computing (RC) systems, especially Field Programmable Gate Arrays (FPGAs), with a focus on the architectural side and the comprehensive software stack: the so-called Domain-Specific Reconfigurable Architectures (DSRAs).

DSRAs represent one of the hottest trends of RC systems (Chapter 2). They can be clustered based on the software programmability and datapath configurability. Their three main classes are software-programmable DSA,

streaming architectures, which leverage design automation toolchains, Coarse Grain Reconfigurable Architecture (CGRA), which are mainly a theoretical platform [38]. On top of their architecture, design and programmability abstractions are essential elements to account in the DSRA methodologies (Chapters 2 and 3) for efficient, usable, and reproducible platforms. Given that each computational domain presents different characteristics (e.g., computational patterns, precision requirements), this dissertation presents domain-specific results of different DSRAs in terms of design methodologies, automation, and usability.

Image Registration (IRG) domain usually leverages context-specific features to decrease monster execution times of optimization procedures at a reasonable power budget. Therefore, this dissertation presents a cross-platform open-source[1] design automation framework that exploits a highly customizable streaming architecture for the most compute-intensive part of IRG, the similarity metric calculation (Chapter 4). This DSRA employs a dataflow map-reduce computational pattern to suit the requirements, the ability to scale to different deployment devices, and a Python-based software layer to deliver state-of-the-art performance and energy efficiencies.

Moving ahead in the considered domains, Regular Expressions (REs) matching is an intrinsically sequential and control-intensive computation that has a significant importance [44, 160, 300, 316]. Nevertheless, its computation nature makes this kernel extremely attractive for spatial architectures [316] and especially for what this dissertation call streaming DSRAs. However, this usually means sacrificing flexibility, which is unacceptable for RE matching. Therefore, this dissertation presents two software-programmable DSAs that exploit the Deterministic Finite Automata (DFA) (Chapter 5) and Non-deterministic Finite Automata (NFA)[2] (Chapter 6) computational approaches to tackle this issue, while still delivering remarkable performance and energy efficiencies.

7.1 Limitations and Future Work

Although this dissertation advocates for the centrality of RC in the *domain specialization*, not every domain and not every volume's size can benefit from reconfigurable systems adoptions. For instance, though it presents several interesting characteristics, the database domain suffers from memory-bandwidth and programmability problems [350]. Nevertheless, the newer generations of reconfigurable systems will become key players in memory-bandwidth hungry applications through the integration of hard-blocks for

[1] https://github.com/necst/iron [2] https://github.com/necst/cicero

interconnectivity, such as OpenCAPI [147], and powerful memories, such as High-Bandwidth Memory (HBM) [351].

On top of this, the increasing heterogeneity, e.g., Xilinx ACAP [67], would undoubtedly increase the performance harvestable from the same reconfigurable system but poses enormous challenges in the hardware/software co-design and increases the complexity of the design itself. To this extent, we think that CGRAs [38] will achieve a key role, thanks to their flexibility and computation capabilities. However, the reconfigurable computing community would have to put a significant effort into extending to CGRAs the work done so far for FPGAs. Along with these architectural and use improvements, Computer Aided Design (CAD) tools and methodologies of software-hardware co-design must grow to make all these great features usable. Any hardware improvement would not lead to further steps without usability, but only new useless great hardware. Moreover, some computational models, such as the bit-serial one [50][3], show to be more effective whenever tightly coupled with the memory locations, such as the cache [352] or the Dynamic RAM (DRAM) itself [353]. Indeed, this so-called Processing In Memory (PIM) usually exploits the bit-serial nature of main memories to bring the computation directly into the memory chip. In this regard, as a future direction, we envision an investigation in this ecosystem that is rapidly growing with exciting results [354, 355].

Then, this dissertation presented DSRAs and their comprehensive stack with their limits and advantages and possible future directions for specific domains. The design automation framework for IRG presented highly encouraging results, with state-of-the-art performance and energy efficiency. Although the streaming DSRA is optimized and delivered with high-level Application Programming Interfaces (APIs), the domain may present peculiarities where other similarity metrics perform better than the Mutual Information (MI), and a complete open-source application layer is missing. Moreover, the variety of computations required forbids the creation of a simple DSA. However, we envision that a heterogeneous CGRA may represent the viable compromise to tackle IRG computation challenges.

Considering the REs domain, there is no a clear computational winner among TiReX (Chapter 5) and CICERO (Chapter 6). Although the REs as instruction methodology showcases remarkable achievements for both the DSAs, the architectures present their limitations. On the one hand, TiReX suffers the algorithmic backtracking issue, and it cannot handle this issue currently. On the other hand, CICERO demonstrates throughput limitations and load balancing problems that limit the engine number scaling.

[3] https://github.com/EECS-NTNU/bismo

Within this context, we envision improvements on the architectural side to tackle both issues. We foresee the creation of co-design approaches where software and hardware automatically collaborate to leverage shared information and dramatically improve the overall process. On top of this, an heterogeneous multi-core architecture combined of TiReX and CICERO core can deliver the requested performance and run-time adaptability to the workloads, similarly to a big.LITTLE architecture [356].

On top of these design methodologies and automation frameworks, a Design Space Exploration (DSE) methodology might improve the design process further while providing interesting insights on utilization and achievable frequency, such as open-source frameworks[4] [53]. Moreover, adding (static) performance models (e.g., Roofline [357]) and approximation models of the physical toolchain run could give an extreme boost to DSRAs development and continuous deployment.

7.2 Dissertation Takeaways

Overall, the main takeaways from this dissertation are the following.

Takeaway 1: RC systems, and especially FPGAs, are and will be central to the *domain-specialization* direction. Their reconfigurable and heterogeneous features make these devices attractive for domain-specialized adaptive architectures.

Takeaway 2: Reconfigurable systems will play a crucial role in cloud computing and High Performance Computing (HPC). Among the ongoing projects, IBM cloudFPGA [134, 135, 358] and MSR Honeycomb [359] envision distributed systems with Central Processing Unit (CPU)-free nodes, where FPGAs would be able to remove the burden of generalities coming from CPUs, or collaborate in a disaggregated fashion [360].

Takeaway 3: The field of digital design abstraction for FPGAs is cyclical. From a very specific language for hardware description (i.e., VHDL and Verilog), we move to High-Level Synthesis (HLS) with the employment of generic C language, and we finally find Domain-Specific Languages (DSLs) with their extreme specificity for given computations.

Takeaway 4: Among DSL clusters in Chapter 3, the intermediate infrastructure one represents an exciting trend for hardware design. Indeed, it may embody the cyclicity in this field and the back to the generality. In this way, researchers could build general-purpose languages that can run not only on CPUs but also on FPGAs [249]. Undoubtedly, this trend will require a specialized compilation and toolchain flow for a hardware design

[4] https://github.com/necst/dovado

that can be vendor-agnostic and open, similar to LLVM [361].

Takeaway 5: CAD or general design automation toolchains along with high-level generators [171] are essential to increase productivity towards iterative lifecycles [14, 15], lower the entry-level barrier, enable large-scale emulation platforms [139], and ease the reproducibility.

Takeaway 6: A usability layer that comprehends automation/generators and software abstractions is paramount for every kind of architecture that would like to be used by someone beyond the developers themself.

Takeaway 7: There is no single architecture able to tackle every domain problem optimally. Each domain presents different characteristics that have to be taken into account for the design of DSRAs. However, this consideration might imply the bloom of DSAs for each domain, which is a non-negligible effort. Therefore, applications needs will drive the real necessity of developing such architectures.

Takeaway 8: A domain might present more than a single computational pattern that fits the requirements of the different workloads. Based on these needs, it is paramount to consider all of them and provide an adaptable architecture. An example is the REs domain, where depending on the degree of alternatives and the dataset, a NFA-like execution mode impacts dramatically against a DFA-like in both positive and negative aspects.

Takeaway 9: Adaptability has a cost. From run-time resource estimations to datapath modifications, from precision variations to computation fitting. All this flexibility imposes an unavoidable tax.

Takeaway 10: Though CGRAs are mainly a theoretical platform [38], they will become more and more relevant. On top of significant abstraction efforts, CGRAs will represent the architecture that fill the gap between stream architectures and DSA for their coarse-grained reconfigurability nature and massive (heterogeneous) parallelism degree.

Takeaway 11: What comes naturally after this dissertation is our belief in a future open FPGA ecosystem. While the software advanced thanks to the open-source world, the hardware community is still stuck in the closed version, without contributing to the ecosystem growth. Similarly to the RISC-V revolution [362, 363], we envision a open ecosystem for FPGAs[5] from the architecture itself [364, 365] to the design toolchains [120, 165]. Indeed, open-source FPGA-based projects are more and more, and even some vendors started to open their toolchains [116, 366, 367].

[5] https://osfpga.org/

List of Acronyms

ACS Accelerator-Centric Synthesis

ALM Adaptive Logic Module

API Application Programming Interface

AP Automata Processor

ASIC Application-Specific Integrated Circuit

ASIP Application-Specific Instruction-set Processor

AWS Amazon Web Services

AXI Advanced eXtensible Interface

BLE Basic Logic Element

BRAM Block RAM

List of Acronyms

BSV BlueSpec SystemVerilog

CAD Computer Aided Design

CGRA Coarse Grain Reconfigurable Architecture

CDR Clock Data Recovery

CLB Configurable Logic Block

CNN Convolutional Neural Network

CPLD Complex Programmable Logic Device

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DDR Double Data Rate Memory

DFA Deterministic Finite Automata

DLL Delay Locked Loop

DNN Deep Neural Network

DPR Dynamic Partial Reconfiguration

DPU Deep Processing Unit

DRAM Dynamic RAM

DSA Domain-Specific Architecture

DSRA Domain-Specific Reconfigurable Architecture

DSL Domain-Specific Language

DSE Design Space Exploration

DSP Digital Signal Processing

EDA Electronic Design Automation

EMIB Embedded Multi-die Interconnect Bridges

FA Finite State Automaton

FF Flip Flop

FPGA Field Programmable Gate Array

FaaS FPGA-as-a-Service

FSM Finite State Machine

GPU Graphics Processing Unit

HaaS Hardware-as-a-Service

HBM High-Bandwidth Memory

HDL Hardware Description Language

List of Acronyms

HLS High-Level Synthesis

HPC High Performance Computing

HPC High Performance Computing

IC Integrated Circuit

IDS Intrusion Detection System

IMA In-Memory Architectures

IR Intermediate Representation

IRG Image Registration

ISA Instruction Set Architecture

LB Logic Block

LUT Look-Up Table

MGT Multi-Gigabit Transceiver

MI Mutual Information

ML Machine Learning

NFA Non-deterministic Finite Automata

NoC Network on Chip

NN Neural Network

NPU Neural Processing Unit

NRE Non-Recurring Engineering

PAL Programmable Array Logic

PD Phase Detector

PEN Power Edge of Network™

PISO Parallel In Serial Out

PIM Processing In Memory

PLL Phase Locked Loop

PLA Programmable Logic Array

QPI QuickPath Interconnect

RAM Random Access Memory

RC Reconfigurable Computing

RE Regular Expression

RMT Reconfigurable Match Tables

RTL Register Transfer Level

List of Acronyms

RX Receiver

SerDes Serializer/Deserializer

SIPO Serial In Parallel Out

SiP System in Package

SDA Stream-Dataflow Architectures

SoC System on Chip

SPA Software-Programmable Architectures

SRAM Static Random Access Memory

TCAM Ternary Content Addressable Memory

TPU Tensor Processing Unit

TX Transmitter

URAM Ultra RAM

VCO Voltage Controlled Oscillator

Bibliography

[1] Karl Rupp. 48 years of microprocessor trend data. https://github.com/karlrupp/microprocessor-trend-data, 2021.

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

[3] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[4] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual international symposium on computer architecture (ISCA)*, pages 365–376. IEEE, 2011.

[5] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach 6th edition*. Elsevier, 2018.

[6] Jeff Dean, David Patterson, and Cliff Young. A new golden age in computer architecture: Empowering the machine-learning revolution. *IEEE Micro*, 38(2):21–29, 2018.

[7] Michael Feldman. Intel slowdown in 10nm fabrication process. https://www.nextplatform.com/2019/05/13/intel-gives-moores-law-a-makeover/, 2019.

[8] John L Hennessy and David A Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, 2019.

[9] Emanuele Del Sozzo. On how to effectively target fpgas from domain specific tools. 2019.

[10] 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.

[11] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

Bibliography

[12] Jason Redgrave, Albert Meixner, Nathan Goulding-Hotta, Artem Vasilyev, and Ofer Shacham. Pixel visual core: Google's fully programmable image vision and ai processor for mobile devices. In *Proc. IEEE Hot Chips Symp.(HCS)*, pages 1–18, 2018.

[13] Microsoft Inc. Project Catapult. `https://www.microsoft.com/en-us/research/project/project-catapult/`.

[14] Yunsup Lee, Andrew Waterman, Henry Cook, Brian Zimmer, Ben Keller, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevtic, Stevo Bailey, Milovan Blagojevic, et al. An agile approach to building risc-v microprocessors. *IEEE Micro*, 36(2):8–20, 2016.

[15] AHA Stanford and Agile Hardware Center. Creating an agile hardware flow. *2019 IEEE Hot Chips 31 Symposium (HCS)*, 2019.

[16] Rick Bahr, Clark Barrett, Nikhil Bhagdikar, Alex Carsello, Ross Daly, Caleb Donovick, David Durst, Kayvon Fatahalian, Kathleen Feng, Pat Hanrahan, et al. Creating an agile hardware design flow. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[17] Emanuele Del Sozzo, Davide Conficconi, Alberto Zeni, Mirko Salaris, Donatella Sciuto, and Marco Domenico Santambrogio. Pushing the level of abstraction of digital system design: a survey on how to program fpgas. *Submitted to ACM Computing Surveys (CSUR)*, 54(1):1–38, 2022.

[18] Eleonora D'Arnese, Davide Conficconi, Marco Domenico Santambrogio, and Donatella Sciuto. Reconfigurable architectures: the shift from general systems to domain specific solutions. In Anupam Chattopadhyay Mohamed M. Sabry Aly, editor, *Emerging Computing: From Devices to Systems. Looking Beyond Moore and Von NEumann*, chapter 15. Springer, Singapore, 2022.

[19] Salil Raje. The future of adaptive computing: The composable data center. `https://forums.xilinx.com/t5/Xilinx-Xclusive-Blog/The-Future-of-Adaptive-Computing-The-Composable-Data-Center/ba-p/1221927`, 2021.

[20] Xilinx. Adaptive computing: Hardware that adapts to your application. `https://xilinx.com/applications/adaptive-computing.html`, 2021.

[21] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[22] André DeHon. The density advantage of configurable computing. *Computer*, 33(4):41–49, 2000.

[23] Zhi Guo, Walid Najjar, Frank Vahid, and Kees Vissers. A quantitative analysis of the speedup factors of fpgas over processors. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 162–170, 2004.

[24] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 26(2):203–215, 2007.

[25] Russell Tessier, Kenneth Pocek, and Andre DeHon. Reconfigurable computing architectures. *Proceedings of the IEEE*, 103(3):332–354, 2015.

[26] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (csuR)*, 34(2):171–210, 2002.

[27] Brahim Betkaoui, David B Thomas, and Wayne Luk. Comparing performance and energy efficiency of fpgas and gpus for high productivity computing. In *2010 International Conference on Field-Programmable Technology*, pages 94–101. IEEE, 2010.

[28] Shijie Zhou, Charalampos Chelmis, and Viktor K Prasanna. High-throughput and energy-efficient graph processing on fpga. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 103–110. IEEE, 2016.

[29] Murad Qasaimeh, Kristof Denolf, Jack Lo, Kees Vissers, Joseph Zambreno, and Phillip H Jones. Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels. In *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, pages 1–8. IEEE, 2019.

[30] Davide Conficconi, Eleonora D'Arnese, Emanuele Del Sozzo, Donatella Sciuto, and Marco D Santambrogio. A framework for customizable fpga-based image registration accelerators. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 251–261, 2021.

[31] Giulia Gerometta, Davide Conficconi, and Marco D Santambrogio. On how FPGAs are changing the computer security panorama: An educational survey. In *2021 IEEE 6th International Forum on Research and Technology for Society and Industry (RTSI) (IEEE RTSI 2021)*, Napoli, Italy, September 2021.

[32] 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 *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24. IEEE, 2014.

[33] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[34] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. Enabling fpgas in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, pages 1–10, 2014.

[35] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, and Song Jiang. Sda: Software-defined accelerator for large-scale dnn systems. In *2014 IEEE Hot Chips 26 Symposium (HCS)*, pages 1–23. IEEE, 2014.

[36] David Pellerin. Fpga accelerated computing using aws f1 instances. *AWS Public Sector Summit*, 2017.

[37] Huawei. Huawei releases the new-generation intelligent cloudhardware platform atla. https://e.huawei.com/us/news/global/2017/201709061557, 2017.

[38] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. *ACM Computing Surveys (CSUR)*, 52(6):1–39, 2019.

[39] Kermin E Fleming, Kent D Glossop, Simon C Steely Jr, Jinjie Tang, Alan G Gara, et al. Processors, methods, and systems with a configurable spatial accelerator, July 5 2018. US Patent App. 15/396,402.

[40] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. [dl] a survey of fpga-based neural network inference accelerators. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 12(1):1–26, 2019.

[41] Stylianos I Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

Bibliography

[42] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. Stream-dataflow acceleration. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 416–429. IEEE, 2017.

[43] Enrico Reggiani, Emanuele Del Sozzo, Davide Conficconi, Giuseppe Natale, Carlo Moroni, and Marco D. Santambrogio. Enhancing the scalability of multi-fpga stencil computations via highly optimized hdl components. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 14(3):1–33, 2021.

[44] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2006-183*, 2006.

[45] Francisco PM Oliveira and Joao Manuel RS Tavares. Medical image registration: a review. *Computer methods in biomechanics and biomedical engineering*, 17(2):73–93, 2014.

[46] Eleonora D'Arnese, GW Di Donato, Emanuele Del Sozzo, and Marco D Santambrogio. Towards an automatic imaging biopsy of non-small cell lung cancer. In *2019 IEEE EMBS International Conference on Biomedical & Health Informatics (BHI)*, pages 1–4. IEEE, 2019.

[47] Kadir Cenk Alpay, Kadir Berkay Aydemir, and Alptekin Temizel. Accelerating translational image registration for hdr images on gpu. *arXiv preprint arXiv:2007.06483*, 2020.

[48] Alessandro Comodi, Davide Conficconi, Alberto Scolari, and Marco D Santambrogio. Tirex: Tiled regular expression matching architecture. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 131–137. IEEE, 2018.

[49] Russ Cox. Regular expression matching can be simple and fast. `http://swtch.com/rsc/regexp/regexp1.html`, 2007.

[50] Yaman Umuroglu, Davide Conficconi, Lahiru Rasnayake, Thomas B Preusser, and Magnus Själander. Optimizing bit-serial matrix multiplication for reconfigurable computing. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 12(3):1–24, 2019.

[51] Davide Conficconi, Emanuele Del Sozzo, Filippo Carloni, Alessandro Comodi, Alberto Scolari, and Marco Domenico Santambrogio. An energy-efficient domain-specific architecture for regular expressions. *Submitted to IEEE Transactions on Emerging Topics in Computing*, 2022.

[52] Daniele Parravicini, Davide Conficconi, Emanuele Del Sozzo, Christian Pilato, and Marco Domenico Santambrogio. Cicero: A domain-specific architecture for efficient regular expression matching. *ACM Transactions on Embedded Computing Systems (TECS) for CASES21*, 20(5s), 2021.

[53] Daniele Paletti, Davide Conficconi, and Marco D Santambrogio. Dovado: An open-source design space exploration framework. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 128–135. IEEE, 2021.

[54] Young-Kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. In-depth analysis on microarchitectures of modern heterogeneous cpu-fpga platforms. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 12(1):1–20, 2019.

[55] Christoforos Kachris and Dimitrios Soudris. A survey on reconfigurable accelerators for cloud computing. In *2016 26th International conference on field programmable logic and applications (FPL)*, pages 1–10. IEEE, 2016.

[56] Antonio De La Piedra, An Braeken, and Abdellah Touhafi. Sensor systems based on fpgas and their applications: A survey. *Sensors*, 12(9):12235–12264, 2012.

[57] Kenneth Pocek, Russell Tessier, and André DeHon. Birth and adolescence of reconfigurable computing: A survey of the first 20 years of field-programmable custom computing machines. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 1–17. IEEE, 2013.

[58] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[59] 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, 2015.

[60] 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.

[61] Xilinx. PYNQ: Python for productivity for zynq. http://www.pynq.io/, 2016.

[62] Kizheppatt Vipin and Suhaib A Fahmy. Fpga dynamic and partial reconfiguration: a survey of architectures, methods, and applications. *ACM Computing Surveys (CSUR)*, 51(4):1–39, 2018.

[63] Ian Kuon, Russell Tessier, Jonathan Rose, et al. Fpga architecture: Survey and challenges. *Foundations and Trends® in Electronic Design Automation*, 2(2):135–253, 2008.

[64] Andrew Boutros and Vaughn Betz. Fpga architecture: Principles and progression. *IEEE Circuits and Systems Magazine*, 21(2):4–29, 2021.

[65] Clive Maxfield. *The design warrior's guide to FPGAs: devices, tools and flows*. Elsevier, 2004.

[66] Altera. Stratix ii device handbook, volume 1 (sii5v1-4.5). `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stx2/stratix2_handbook.pdf`, 2007. Last accessed 5 October 2021.

[67] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. Xilinx adaptive compute acceleration platform: Versaltm architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 84–93, 2019.

[68] Jeffrey Chromczak, Mark Wheeler, Charles Chiasson, Dana How, Martin Langhammer, Tim Vanderhoek, Grace Zgheib, and Ilya Ganusov. Architectural enhancements in intel® agilex™ fpgas. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 140–149, 2020.

[69] David Lewis, Vaughn Betz, David Jefferson, Andy Lee, Chris Lane, Paul Leventis, Sandy Marquardt, Cameron McClintock, Bruce Pedersen, Giles Powell, et al. The stratixπ routing and logic architecture. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 12–20, 2003.

[70] Maxeler Technologies. Mpc-x series. `https://www.maxeler.com/products/mpc-xseries/`, 2015.

[71] A Theodore Markettos, Paul J Fox, Simon W Moore, and Andrew W Moore. Interconnect for commodity fpga clusters: Standardized or customized? In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2014.

[72] Altera. Implementing RAM functions in FLEX 10K Devices (A-AN-052-01). `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/archives/an052_01.pdf`, 1995. Last accessed 15 August 2021.

Bibliography

[73] Kosuke Tatsumura, Sadegh Yazdanshenas, and Vaughn Betz. High density, low energy, magnetic tunnel junction based block rams for memory-rich fpgas. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 4–11. IEEE, 2016.

[74] Tony Ngai, Jonathan Rose, and Steven JE Wilton. An sram-programmable field-configurable memory. In *Proceedings of the IEEE 1995 Custom Integrated Circuits Conference*, pages 499–502. IEEE, 1995.

[75] Steven JE Wilton, Jonathan Rose, and Zvonko G Vranesic. Architecture of centralized field-configurable memory. In *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, pages 97–103, 1995.

[76] Sadegh Yazdanshenas, Kosuke Tatsumura, and Vaughn Betz. Don't forget the memory: Automatic block ram modelling, optimization, and architecture exploration. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 115–124, 2017.

[77] Xilinx. Virtex-II Platform FPGAs: Complete Data Sheet. `https://www.xilinx.com/support/documentation/data_sheets/ds031.pdf`, 2014. Last accessed 15 August 2021.

[78] Martin Langhammer, Eriko Nurvitadhi, Bogdan Pasca, and Sergey Gribok. Stratix 10 nx architecture and applications. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 57–67, 2021.

[79] Achronix Semiconductor Corporation. Speedster7t Machine Learning Processing User Guide (UG088). `https://www.achronix.com/sites/default/files/docs/Speedster7t_Machine_Learning_Processor_User_Guide_UG088.pdf`, 2019. Last accessed 15 August 2021.

[80] Ian Swarbrick, Dinesh Gaitonde, Sagheer Ahmad, Brian Gaide, and Ygal Arbel. Network-on-chip programmable platform in versaltm acap architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 212–221, 2019.

[81] Achronix Semiconductor Corporation. Speedster7t Network on Chip User Guide (UG089) . `https://tinyurl.com/achronixnoc`. Last accessed 15 June 2021.

[82] Raghunandan Chaware, Kumar Nagarajan, and Suresh Ramalingam. Assembly and reliability challenges in 3d integration of 28nm fpga die on a large high density 65nm passive interposer. In *2012 IEEE 62nd Electronic Components and Technology Conference*, pages 279–283. IEEE, 2012.

[83] Sergey Shumarayev. Stratix 10: Intel's 14nm heterogeneous fpga system-in-package (sip) platform. In *Proc. Hot Chips 29 Symp.*, 2017.

[84] Stephen M Steve Trimberger. Three ages of fpgas: A retrospective on the first thirty years of fpga technology: This paper reflects on how moore's law has driven the design of fpgas through three epochs: the age of invention, the age of expansion, and the age of accumulation. *IEEE Solid-State Circuits Magazine*, 10(2):16–29, 2018.

[85] CUDA NVIDIA. Compute unified device architecture programming guide. *Nvidia website*, 2007.

[86] Louise H Crockett, Ross A Elliot, Martin A Enderwitz, and Robert W Stewart. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014.

[87] Neal Oliver, Rahul R Sharma, Stephen Chang, Bhushan Chitlur, Elkin Garcia, Joseph Grecco, Aaron Grier, Nelson Ijih, Yaping Liu, Pratik Marolia, et al. A reconfigurable computing system based on a cache-coherent fabric. In *2011 International Conference on Reconfigurable Computing and FPGAs*, pages 80–85. IEEE, 2011.

[88] Dimitrios Ziakas, Allen Baum, Robert A Maddox, and Robert J Safranek. Intel® quickpath interconnect architectural features supporting scalable system architectures. In *2010 18th IEEE Symposium on High Performance Interconnects*, pages 1–6. IEEE, 2010.

[89] Shant Chandrakar, Dinesh Gaitonde, and Trevor Bauer. Enhancements in ultrascale clb architecture. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 108–116, 2015.

[90] Lesley Shannon, Veronica Cojocaru, Cong Nguyen Dao, and Philip HW Leong. Technology scaling in fpgas: Trends in applications and architectures. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 1–8. IEEE, 2015.

[91] Tom Feist. Vivado design suite. *White Paper*, 5:30, 2012.

[92] Intel. Intel quartus documentation. https://www.intel.com/content/www/us/en/programmable/products/design-software/fpga-design/quartus-prime/user-guides.html, 2020.

[93] 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`, 2017.

[94] Tomasz S Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P Singh. From opencl to high-performance hardware on fpgas. In *22nd international conference on field programmable logic and applications (FPL)*, pages 531–534. IEEE, 2012.

[95] Alter. Implementing fpga design with theopencl standard. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01173-opencl.pdf, 2013.

[96] Xilinx. Sdaccel press release. https://www.xilinx.com/news/press/2014/xilinx-announces-sdaccel-development-environment-for-opencl-c-and-c–delivering-up-to-25x-better-performance-watt-to-the-data-center.html, 2014.

[97] Xilinx. Xilinx vitis unified software platform. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html, October 2019.

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

[99] 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, 2011.

[100] 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 *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 619–622. IEEE, 2012.

[101] David Koeplinger, Raghu Prabhakar, Yaqi Zhang, Christina Delimitrou, Christos Kozyrakis, and Kunle Olukotun. Automatic generation of efficient accelerators for reconfigurable hardware. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 115–127. Ieee, 2016.

[102] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 389–402. IEEE, 2017.

Bibliography

[103] David Grant, Chris Wang, and Guy GF Lemieux. A cad framework for malibu: An fpga with time-multiplexed coarse-grained elements. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 123–132, 2011.

[104] Florian Fricke, André Werner, Keyvan Shahin, and Michael Hübner. Cgra tool flow for fast run-time reconfiguration. In *International Symposium on Applied Reconfigurable Computing*, pages 661–672. Springer, 2018.

[105] S Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. Cgra-me: A unified framework for cgra modelling and exploration. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 184–189. IEEE, 2017.

[106] Richard M Karp, Raymond E Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM (JACM)*, 14(3):563–590, 1967.

[107] Moritz Schmid, Frank Hannig, Ru Tanase, and Jürgen Teich. High-level synthesis revised: Generation of fpga accelerators from a domain-specific language using the polyhedron model. 2013.

[108] 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.

[109] Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong, and Jason Cong. Improving polyhedral code generation for high-level synthesis. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 1–10. IEEE, 2013.

[110] Giuseppe Natale, Giulio Stramondo, Pietro Bressana, Riccardo Cattaneo, Donatella Sciuto, and Marco D Santambrogio. A polyhedral model-based framework for dataflow implementation on fpga devices of iterative stencil loops. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2016.

[111] Jongsok Choi, Stephen D Brown, and Jason H Anderson. From pthreads to multicore hardware systems in legup high-level synthesis for fpgas. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2867–2880, 2017.

[112] Marco Rabozzi, Rolando Brondolin, Giuseppe Natale, Emanuele Del Sozzo, Michael Huebner, Andreas Brokalakis, Catalin Ciobanu, Dirk Stroobandt, and Marco Domenico Santambrogio. A cad open platform for high performance reconfigurable systems in the extra project. In *2017 IEEE computer society annual symposium on VLSI (ISVLSI)*, pages 368–373. IEEE, 2017.

[113] Lorenzo Di Tucci, Marco Rabozzi, Luca Stornaiuolo, and Marco D Santambrogio. The role of cad frameworks in heterogeneous fpga-based cloud systems. In *2017 IEEE international conference on computer design (ICCD)*, pages 423–426. IEEE, 2017.

[114] Marco Rabozzi, Giuseppe Natale, Emanuele Del Sozzo, Alberto Scolari, Luca Stornaiuolo, and Marco D Santambrogio. Heterogeneous exascale supercomputing: the role of cad in the exafpga project. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 410–415. IEEE, 2017.

[115] Christopher Lavin, Marc Padilla, Jaren Lamprecht, Philip Lundrigan, Brent Nelson, and Brad Hutchings. Rapidsmith: Do-it-yourself cad tools for xilinx fpgas. In *2011 21st International Conference on Field Programmable Logic and Applications*, pages 349–355. IEEE, 2011.

[116] Chris Lavin and Alireza Kaviani. Rapidwright: Enabling custom crafted implementations for fpgas. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 133–140. IEEE, 2018.

[117] Symbiflow. Symbiflow project website. `https://symbiflow.github.io/`, 2018.

[118] IceStorm. Project icestorm website. `http://www.clifford.at/icestorm/`, 2015.

[119] Project X-Ray. Project x-ray repository. `https://github.com/SymbiFlow/prjxray`, 2017.

[120] David Shah, Eddie Hung, Clifford Wolf, Serge Bazanski, Dan Gisselquist, and Miodrag Milanovic. Yosys+ nextpnr: an open source framework from verilog to bitstream for commercial fpgas. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–4. IEEE, 2019.

[121] Deshanand Singh. Implementing fpga design with the opencl standard. *Altera whitepaper*, 1, 2011.

[122] Loring Wirbel. Xilinx sdaccel: a unified development environment for tomorrow's data center. *The Linley Group Inc*, 2014.

[123] Maxeler. Maxcompiler white paper. https://www.maxeler.com/media/documents/MaxelerWhitePaperMaxCompiler.pdf, 2011.

[124] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.

[125] Emanuele Del Sozzo, Riyadh Baghdadi, Saman Amarasinghe, and Marco D Santambrogio. A common backend for hardware acceleration on fpga. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 427–430. IEEE, 2017.

[126] 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):1–25, 2017.

[127] Sean O Settle et al. High-performance dynamic programming on fpgas with opencl. In *Proc. IEEE High Perform. Extreme Comput. Conf.(HPEC)*, pages 1–6, 2013.

[128] Jack B Dennis. Data flow supercomputers. *Computer*, (11):48–56, 1980.

[129] Google. Experiences building edge tpu with chisel. https://www.youtube.com/watch?v=x85342Cny8c, 2018.

[130] Steven Margerm, Amirali Sharifian, Apala Guha, Arrvindh Shriraman, and Gilles Pokam. Tapas: Generating parallel accelerators from parallel programs. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 245–257. IEEE, 2018.

[131] 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.

[132] Jessica Morgan Ray. *A unified compiler backend for distributed, cooperative heterogeneous execution*. PhD thesis, Massachusetts Institute of Technology, 2018.

[133] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205. IEEE, 2019.

Bibliography

[134] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner, and Andreas Herkersdorf. Enabling fpgas in hyperscale data centers. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, pages 1078–1086. IEEE, 2015.

[135] Francois Abel, Jagath Weerasinghe, Christoph Hagleitner, Beat Weiss, and Stephan Paredes. An fpga platform for hyperscalers. In *2017 IEEE 25th Annual Symposium on High-Performance Interconnects (HOTI)*, pages 29–32. IEEE, 2017.

[136] Mohammad Hosseinabady and José Luis Núñez-Yáñez. Pipelined streaming computation of histogram in fpga opencl. In *PARCO*, pages 632–641, 2017.

[137] Vinh Dang and Kevin Skadron. Acceleration of frequent itemset mining on fpga using sdaccel and vivado hls. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 195–200. IEEE, 2017.

[138] Gabriel S Niemiec, Luis MS Batista, Alberto E Schaeffer-Filho, and Gabriel L Nazar. A survey on fpga support for the feasible execution of virtualized network functions. *IEEE Communications Surveys & Tutorials*, 2019.

[139] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 29–42. IEEE, 2018.

[140] Mia Champion. Bringing datacenter-scale hardware-software co-design to the cloud with firesim and amazon ec2 f1 instances. *AWS Compute Blog*, 2017.

[141] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. A quantitative analysis on microarchitectures of modern cpu-fpga platforms. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[142] AlphaData. Alphadata reconfigurable computing for hpc boards. https://www.alpha-data.com/dcp/, 2020.

[143] Bruce Wile. Coherent accelerator processor interface (capi) for power8 systems white paper. *IBM Systems and Technology Group*, 2014.

[144] PK Gupta. Accelerating datacenter workloads. In *26th International Conference on Field Programmable Logic and Applications (FPL)*, volume 2017, page 20, 2016.

[145] OpenCAPI consortium. Tech leaders unite to enable new cloud datacenter server designs for big data, machine learning, analytics, and other emerging workloads. https://opencapi.org/2016/10/tech-leaders-unite-to-enable-new-cloud-datacenter-server-designs-for-big-data-machine-learning-analytics-and-other-emerging-workloads/, October 2016.

[146] CCIX. Ccix: a new coherent multichip interconnect for accelerated use cases. https://www.ccixconsortium.com/wp-content/uploads/2018/08/ArmTechCon17-CCIX-A-New-Coherent-Multichip-Interconnect-for-Accelerated-Use-Cases.pdf, 2017.

[147] OpenCAPI. Opencapi a data-centric approach to server design. https://opencapi.org/wp-content/uploads/2016/09/OpenCAPI-Exhibit-SC17.pdf, 2016.

[148] Anupam Chattopadhyay, Rainer Leupers, Heinrich Meyr, and Gerd Ascheid. *Language-driven exploration and implementation of partially re-configurable ASIPs*. Springer Science & Business Media, 2008.

[149] Francisco Barat, Rudy Lauwereins, and Geert Deconinck. Reconfigurable instruction set processors from a hardware/software perspective. *IEEE Transactions on Software Engineering*, 28(9):847–862, 2002.

[150] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Song Yao, Song Han, Yu Wang, and Huazhong Yang. From model to fpga: Software-hardware co-design for efficient neural network acceleration. In *2016 IEEE Hot Chips 28 Symposium (HCS)*, pages 1–27. IEEE, 2016.

[151] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38(2):8–20, 2018.

[152] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 65–74. ACM, 2017.

[153] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 152–159. IEEE, 2017.

[154] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[155] Pavel Benácek, Viktor Pu, and Hana Kubátová. P4-to-vhdl: Automatic generation of 100 gbps packet parsers. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 148–155. IEEE, 2016.

[156] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*, pages 122–135, 2017.

[157] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. Flowblaze: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, Boston, MA, February 2019. USENIX Association.

[158] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[159] Ted Xie et al. Reapr: Reconfigurable engine for automata processing. In *Proc. of FPL*, pages 1–8, 2017.

[160] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. Flexamata: A universal and efficient adaption of applications to spatial automata processing accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 219–234, 2020.

[161] Jason Cong, Vivek Sarkar, Glenn Reinman, and Alex Bui. Customizable domain-specific computing. *IEEE Design & Test of Computers*, 28(2):6–15, 2010.

[162] Xilinx Inc. Vivado Design Suite. `http://www.xilinx.com/products/design-tools/vivado.html`, 2013.

Bibliography

[163] Synopsys. RTL Synhtesis and Test. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test.html, 2021.

[164] Mentor Graphics. Design Creation. https://www.mentor.com/products/fpga/hdl_design/, 2021.

[165] Kevin E Murray, Mohamed A Elgammal, Vaughn Betz, Tim Ansell, Keith Rothman, and Alessandro Comodi. Symbiflow and vpr: An open-source design flow for commercial and novel fpgas. *IEEE Micro*, 40(4):49–57, 2020.

[166] Samir Palnitkar. *Verilog HDL: a guide to digital design and synthesis*, volume 1. Prentice Hall Professional, 2003.

[167] Michael D Ciletti. *Advanced digital design with the Verilog HDL*, volume 1. Prentice Hall Upper Saddle River, 2003.

[168] Ricardo Jasinski. *Effective coding with VHDL: principles and best practice*. MIT Press, 2016.

[169] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[170] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2004.

[171] The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[172] Derek Lockhart, Stephen Twigg, Ravi Narayanaswami, Jeremy Coriell, Uday Dasari, Richard Ho, Doug Hogberg, George Huang, Anand Kane, Chintan Kaur, Tao Liu, Adriana Maggiore, Kevin Townsend, and Emre Tuncer. Experiences building edge tpu with chisel. https://www.youtube.com/watch?v=x85342Cny8c, 2018.

[173] Sanjit Seshia, Albert Magyar, David Biancolin, John Koenig, Jonathan Bachrach, and Krste Asanovic. Golden gate: Bridging the resource-efficiency gap between asics and fpga prototypes. 2021.

[174] Jan Decaluwe. Myhdl: a python-based hardware description language. *Linux journal*, (127):84–87, 2004.

[175] John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahan, and Timothy Sherwood. A pythonic approach for rapid hardware prototyping and instrumentation. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–7. IEEE, 2017.

[176] 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.

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

[178] Maxeler Inc. Multiscale dataflow programming. https://www.maxeler.com/products/software/maxcompiler/, 2021.

[179] David I Rich. The evolution of systemverilog. *IEEE Annals of the History of Computing*, 20(04):82–84, 2003.

[180] 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.

[181] Rishiyur Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.*, pages 69–70. IEEE, 2004.

[182] Thomas Bourgeat, Clément Pit-Claudel, and Adam Chlipala. The essence of bluespec: a core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 243–257, 2020.

[183] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. Cλash: Structural descriptions of synchronous hardware using haskell. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, pages 714–721. IEEE, 2010.

[184] C Papon. Spinalhdl: An alternative hardware description language. *FOSDEM*, 2017.

[185] Oron Port and Yoav Etsion. Dfiant: A dataflow hardware description language. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017.

[186] Yanqiang Liu, Yao Li, Zhengwei Qi, and Haibing Guan. A scala based framework for developing acceleration systems with fpgas. *Journal of Systems Architecture*, 98:231–242, 2019.

[187] Olav Lindtjorn, Robert G Clapp, Oliver Pell, Oskar Mencer, and Michael J Flynn. Surviving the end of scaling of traditional microprocessors in hpc. *IEEE Hot Chips*, 22:22–24, 2010.

[188] Simpei Sato and Kenji Kise. Archhdl: a new hardware description language for high-speed architectural evaluation. In *2013 IEEE 7th International Symposium on Embedded Multicore Socs*, pages 107–112. IEEE, 2013.

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

[190] Steven F Hoover. Timing-abstract circuit design in transaction-level verilog. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 525–532. IEEE, 2017.

[191] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems*, 16(3):31–51, 2012.

[192] Philippe Coussy, Daniel D Gajski, Michael Meredith, and Andres Takach. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, 2009.

[193] Sakari Lahti, Panu Sjövall, Jarno Vanne, and Timo D Hämäläinen. Are we there yet? a study on the state of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5):898–911, 2018.

[194] Grant Martin and Gary Smith. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009.

[195] Silexica. SLX FPGA. `https://www.silexica.com/products/slx-fpga/`, 2021.

[196] M. Rabozzi, G. Natale, E. Del Sozzo, A. Scolari, L. Stornaiuolo, and M. D. Santambrogio. Heterogeneous exascale supercomputing: The role of cad in the exafpga project. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 410–415, 2017.

[197] Jie Wang, Licheng Guo, and Jason Cong. Autosa: A polyhedral compiler for high-performance systolic arrays on fpga. In *Proceedings of the 2021 ACM/SIGDA international symposium on Field-programmable gate arrays*, 2021.

[198] Xilinx Inc. Vivado HLS. `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`, 2013.

Bibliography

[199] Xilinx Inc. Introduction to Vitis HLS. https://www.xilinx.com/html_docs/xilinx2020_1/vitis_doc/introductionvitishls.html, 2020.

[200] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. Invited tutorial: Dynamatic: From c/c++ to dynamically scheduled circuits. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 1–10, 2020.

[201] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. DYNAMATIC - Dynamically Scheduled High-Level Synthesis. https://github.com/lana555/dynamatic, 2019.

[202] Intel. Intel FPGA SDK for OpenCL. https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html, 2021.

[203] Tao B Schardl, William S Moses, and Charles E Leiserson. Tapir: Embedding fork-join parallelism into llvm's intermediate representation. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 249–265, 2017.

[204] Xilinx Inc. Vitis Unified Software Platform. https://www.xilinx.com/products/design-tools/vitis.html, 2019.

[205] Xilinx Inc. SDAccel Development Environment. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html, 2019.

[206] Xilinx Inc. SDSoC Development Environment. https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html, 2018.

[207] Xilinx Inc. Vitis Accelerated Libraries. https://github.com/Xilinx/Vitis_Libraries, 2021.

[208] Xilinx Inc. Xilinx Runtime Library. https://github.com/Xilinx/XRT, 2021.

[209] MathWorks Inc. HDL Coder User's Guide. https://www.mathworks.com/help/pdf_doc/hdlcoder/hdlcoder_ug.pdf.

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

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

[212] Satnam Singh and David J Greaves. Kiwi: Synthesis of fpga circuits from parallel programs. In *2008 16th International Symposium on Field-Programmable Custom Computing Machines*, pages 3–12. IEEE, 2008.

[213] 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.

[214] 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.

[215] NEC. CyberWorkBench: High Level Synthesis from C/C++/SystemC to ASIC/FPGA. https://www.nec.com/en/global/prod/cwb/index.html, 2011.

[216] 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.

[217] Cadence. Impulse C User Guide. https://www.cadence.com/ko_KR/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html, 2021.

[218] 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.

[219] Lombiq Technologies. Hastlayer SDK - GitHub. `https://github.com/Lombiq/Hastlayer-SDK`, 2019.

[220] 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`, 2021.

[221] Google. XLS: Accelerated HW Synthesis. `https://google.github.io/xls/`, 2021.

[222] Intel Inc. Intel oneAPI: A Unified X-Architecture Programming Model. `https://software.intel.com/content/www/us/en/develop/tools/oneapi.html`, 2021.

[223] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[224] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan, et al. Warehouse-scale video acceleration: co-design and deployment in the wild. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 600–615, 2021.

[225] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[226] Jakub Cabal, Pavel Benáček, Lukáš Kekely, Michal Kekely, Viktor Puš, and Jan Kořenek. Configurable fpga packet parser for terabit networks with guaranteed wire-speed throughput. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 249–258, 2018.

[227] 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.

[228] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–311, 2018.

[229] The Stanford Pervasive Parallelism Lab. Spatial: Specify Parameterized Accelerators Through Inordinately Abstract Language. `https://github.com/stanford-ppl/spatial`.

[230] Bruno Bodin, Luigi Nardi, M Zeeshan Zia, Harry Wagstaff, Govind Sreekar Shenoy, Murali Emani, John Mawer, Christos Kotselidis, Andy Nisbet, Mikel Lujan, et al. Integrating algorithmic parameters into benchmarking and design space exploration in 3d scene understanding. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 57–69, 2016.

[231] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. A compiler infrastructure for accelerator generators. 2021.

[232] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, et al. Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 209–216. IEEE, 2017.

Bibliography

[233] Richard Membarth, Oliver Reiche, Özkan Mehmet Akif, and Bo Qiao. Repo HIP^{acc}, 2013. last accessed March 31, 2021.

[234] Richard Membarth, Oliver Reiche, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. Hipa^{cc}: A domain-specific language and compiler for image processing. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):210–224, 2015.

[235] James Hegarty, Ross Daly, Zachary DeVito, Jonathan Ragan-Kelley, Mark Horowitz, and Pat Hanrahan. Rigel: Flexible multi-rate image processing hardware. *ACM Trans. Graph.*, 35(4), July 2016.

[236] Robert Stewart, Kirsty Duncan, Greg Michaelson, Paulo Garcia, Deepayan Bhowmik, and Andrew M. Wallace. RIPL: A Parallel Image Processing Language for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems*, 11(1):7:1–7:24, 2018.

[237] Robert Stewart, Greg Michaelson, Deepayan Bhowmik, Paulo Garcia, and Andy Wallace. Repo RIPL. `https://github.com/robstewart57/ripl`, 2018. last accessed March 31, 2021.

[238] Jeferson Santiago da Silva, François-Raymond Boyer, and JM Pierre Langlois. P4-compatible high-level synthesis of low latency 100 gb/s streaming packet parsers in fpgas. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 147–152, 2018.

[239] Jiajie Li, Yuze Chi, and Jason Cong. Heterohalide: From image processing dsl to efficient fpga acceleration. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 51–57, 2020.

[240] Yi-Hsiang Lai, Hongbo Rong, Size Zheng, Weihao Zhang, Xiuping Cui, Yunshan Jia, Jie Wang, Brendan Sullivan, Zhiru Zhang, Yun Liang, et al. Susy: a programming model for productive construction of high-performance systolic arrays on fpgas. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2020.

[241] Yutaka Watanabe, Jinpil Lee, Kentaro Sano, Taisuke Boku, and Mitsuhisa Sato. Design and preliminary evaluation of openacc compiler for fpga with opencl and stream processing dsl. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Workshops*, pages 10–16, 2020.

[242] Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J Brown, Arvind K Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthesis from domain-specific languages. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2014.

[243] Raghu Prabhakar, David Koeplinger, Kevin J Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. Generating configurable hardware from parallel patterns. *Acm Sigplan Notices*, 51(4):651–665, 2016.

[244] Emanuele Del Sozzo, Riyadh Baghdadi, Saman Amarasinghe, and Marco 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. IEEE, 2018.

[245] Martin Kristien, Bruno Bodin, Michel Steuwer, and Christophe Dubach. High-level synthesis of functional patterns with lift. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, pages 35–45, 2019.

[246] M Akif Özkan, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, Roland Leißa, Sebastian Hack, Jürgen Teich, and Frank Hannig. Anyhls: High-level synthesis with partial evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3202–3214, 2020.

[247] Maya Gokhale and Lesley Shannon. Fpga computing. *IEEE Micro*, 41(4):6–7, 2021.

[248] Kenneth C Louden and Kenneth A Lambert. *Programming languages: principles and practices*. Cengage Learning, 2011.

[249] Amirali Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha, Tony Nowatzki, and Arrvindh Shriraman. μir-an intermediate representation for transforming and optimizing the microarchitecture of application accelerators. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 940–953, 2019.

[250] Lisa Gottesfeld Brown. A survey of image registration techniques. *ACM Computing Surveys (CSUR)*, 24(4):325–376, 1992.

[251] Petra A Van den Elsen, E-JD Pol, and Max A Viergever. Medical image matching-a review with classification. *IEEE Engineering in Medicine and Biology Magazine*, 12(1):26–39, 1993.

[252] James D Foley, Foley Dan Van, Andries Van Dam, Steven K Feiner, John F Hughes, Edward Angel, and J Hughes. *Computer graphics: principles and practice*, volume 12110. Addison-Wesley Professional, 1996.

[253] Ramtin Shams, Parastoo Sadeghi, Rodney A Kennedy, and Richard I Hartley. A survey of medical image registration on multicore and the GPU. *IEEE Signal Processing Magazine*, 27(2):50–60, 2010.

[254] Martin Styner, Christian Brechbuhler, G Szckely, and Guido Gerig. Parametric estimate of intensity inhomogeneities applied to mri. *IEEE transactions on medical imaging*, 19(3):153–165, 2000.

[255] Paul Viola and William M Wells III. Alignment by maximization of mutual information. *International Journal of Computer Vision*, 24(2):137–154, 1997.

[256] Colin Studholme. *Measures of 3D medical image alignment*. PhD thesis, University of London, 1997.

[257] Eleonora D'Arnese, Emanuele Del Sozzo, Davide Conficconi, and Marco D. Santambrogio. Exploiting heterogeneous architectures for rigid image registration. In *2021 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pages 1–4. IEEE, 2021.

[258] Anton Bardera, Miquel Feixas, Imma Boada, Jaume Rigau, and Mateu Sbert. Medical image registration based on BSP and quad-tree partitioning. In *International Workshop on Biomedical Image Registration*, pages 1–8. Springer, 2006.

[259] Ioannis Stratakos, Dimitrios Gourounas, Vasileios Tsoutsouras, Theodore Economopoulos, George Matsopoulos, and Dimitrios Soudris. Hardware acceleration of image registration algorithm on FPGA-based systems on chip. In *Proceedings of the International Conference on Omni-Layer Intelligent Systems*, pages 92–97, 2019.

[260] Davide Conficconi, Eleonora D'Arnese, Emanuele Del Sozzo, Donatella Sciuto, and Marco D. Santambrogio. A framework for customizable fpga-based image registration accelerators. https://github.com/necst/iron, 2020.

[261] Xiyang Zhi, Junhua Yan, Yiqing Hang, and Shunfei Wang. Realization of CUDA-based real-time registration and target localization for high-resolution video images. *Journal of Real-Time Image Processing*, 16(4):1025–1036, 2019.

[262] Taejung Kim and Yong-Jo Im. Automatic satellite image registration by combination of matching and random sample consensus. *IEEE Transactions on Geoscience and Remote Sensing*, 41(5):1111–1117, 2003.

[263] Youcef Bentoutou, Nasreddine Taleb, Kidiyo Kpalma, and Joseph Ronsin. An automatic image registration for applications in remote sensing. *IEEE Transactions on Geoscience and Remote Sensing*, 43(9):2127–2137, 2005.

Bibliography

[264] Frederik Maes, Andre Collignon, Dirk Vandermeulen, Guy Marchal, and Paul Suetens. Multimodality image registration by maximization of mutual information. *IEEE Transactions on Medical Imaging*, 16(2):187–198, 1997.

[265] Flavio Lichtenstein, Fernando Antoneli, and Marcelo RS Briones. Mia: Mutual information analyzer, a graphic user interface program that calculates entropy, vertical and horizontal mutual information of molecular sequence sets. *BMC Bioinformatics*, 16(1):409, 2015.

[266] Atul J Butte and Isaac S Kohane. Mutual information relevance networks: functional genomic clustering using pairwise entropy measurements. In *Biocomputing 2000*, pages 418–429. World Scientific, 1999.

[267] Lalit Bahl, Peter Brown, Peter De Souza, and Robert Mercer. Maximum mutual information estimation of hidden markov model parameters for speech recognition. In *ICASSP'86. IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 11, pages 49–52. IEEE, 1986.

[268] Pablo A Estévez, Michel Tesmer, Claudio A Perez, and Jacek M Zurada. Normalized mutual information feature selection. *IEEE Transactions on Neural Networks*, 20(2):189–201, 2009.

[269] Thomas M Cover and Joy A Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[270] Derek LG Hill and David J Hawkes. Across-modality registration using intensity-based cost functions. *Handbook of Medical Imaging: Processing and Analysis*, pages 537–553, 2000.

[271] Bradley Christopher Lowekamp, David T Chen, Luis Ibáñez, and Daniel Blezek. The design of simpleitk. *Frontiers in neuroinformatics*, 7:45, 2013.

[272] Ziv Yaniv, Bradley C Lowekamp, Hans J Johnson, and Richard Beare. Simpleitk image-analysis notebooks: a collaborative environment for education and reproducible research. *Journal of digital imaging*, 31(3):290–303, 2018.

[273] Inc MathWorks. Matlab, image processing toolbox. `https://mathworks.com/products/image.html`, 1994-2020.

[274] Omkar Dandekar and Raj Shekhar. FPGA-accelerated deformable image registration for improved target-delineation during CT-guided interventions. *IEEE Transactions on Biomedical Circuits and Systems*, 1(2):116–127, 2007.

[275] Carlos R Castro-Pareja, Jogikal M Jagadeesh, and Raj Shekhar. Fair: a hardware architecture for real-time 3-D image registration. *IEEE Transactions on Information Technology in Biomedicine*, 7(4):426–434, 2003.

[276] Omkar Dandekar, William Plishker, Shuvra Bhattacharyya, and Raj Shekhar. Multiobjective optimization of FPGA-based medical image registration. In *2008 16th International Symposium on Field-Programmable Custom Computing Machines*, pages 183–192. IEEE, 2008.

[277] Shifu Chen, Jing Qin, Yongming Xie, Wai-Man Pang, and Pheng-Ann Heng. CUDA-based acceleration and algorithm refinement for volume image registration. In *2009 International Conference on Future BioMedical Information Engineering (FBIE)*, pages 544–547. IEEE, 2009.

[278] Ramtin Shams, Parastoo Sadeghi, Rodney Kennedy, and Richard Hartley. Parallel computation of mutual information on the gpu with application to real-time registration of 3d medical images. *Computer methods and programs in biomedicine*, 99(2):133–146, 2010.

[279] Kei Ikeda, Fumihiko Ino, and Kenichi Hagihara. Efficient acceleration of mutual information computation for nonrigid registration using CUDA. *IEEE Journal of Biomedical and Health Informatics*, 18(3):956–968, 2014.

[280] Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer. Parallel programming for FPGAs. *arXiv preprint arXiv:1805.03648*, 2018.

[281] Kenneth Clark, Bruce Vendt, Kirk Smith, John Freymann, Justin Kirby, Paul Koppel, Stephen Moore, Stanley Phillips, David Maffitt, Michael Pringle, et al. The cancer imaging archive (tcia): maintaining and operating a public information repository. *Journal of digital imaging*, 26(6):1045–1057, 2013.

[282] National Cancer Institute Clinical Proteomic Tumor Analysis Consortium (CPTAC). Radiology data from the clinical proteomic tumor analysis consortium lung adenocarcinoma [cptac-luad] collection [data set]. *The Cancer Imaging Archive*, 2018.

[283] J. Wadden, T. Tracy, E. Sadredini, L. Wu, C. Bo, J. Du, Y. Wei, J. Udall, M. Wallace, M. Stan, and K. Skadron. AutomataZoo: A modern automata processing benchmark suite. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 13–24, 2018.

[284] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. Hare: Hardware accelerator for regular expressions. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[285] Chad R Meiners, Jignesh Patel, Eric Norige, Eric Torng, and Alex X Liu. Fast regular expression matching using small tcams for network intrusion detection and prevention systems. In *Proceedings of the 19th USENIX conference on Security*. USENIX Association, 2010.

[286] Jonathan AP Marpaung, M Agni Catur Bhakti, and Setiadi Yazid. A study on application layer classification for firewalls using regular expression matching. In *Advanced Computer Science Applications and Technologies (ACSAT), 2013 International Conference on*. IEEE, 2013.

[287] Shreyas G Singapura, Yi-Hua E. Yang, Anand Panangadan, Tamas Nemeth, and Viktor K. Prasanna. FPGA based accelerator for pattern matching in YARA framework. Technical report, CE, Los Angeles, CA, 2015.

[288] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. Achieving 100gbps intrusion prevention on a single server. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1083–1100, 2020.

[289] VirusTotal. Yara's documentation. `https://yara.readthedocs.io/en/v3.7.0/`, 2016.

[290] Jack Wadden, Vinh Dang, Nathan Brunelle, Tommy Tracy II, Deyuan Guo, Elaheh Sadredini, Ke Wang, Chunkun Bo, Gabriel Robins, Mircea Stan, et al. Anmlzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2016.

[291] Keira Zhou, Jack Wadden, Jeffrey J Fox, Ke Wang, Donald E Brown, and Kevin Skadron. Regular expression acceleration on the micron automata processor: Brill tagging as a case study. In *Big Data (Big Data), 2015 IEEE International Conference on*. IEEE, 2015.

[292] Zsolt István, David Sidler, and Gustavo Alonso. Runtime parameterizable regular expression operators for databases. In *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*, pages 204–211. IEEE, 2016.

[293] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. Accelerating pattern matching queries in hybrid cpu-fpga architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 403–415. ACM, 2017.

Bibliography

[294] Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P Markatos, and Sotiris Ioannidis. Regular expression matching on graphics hardware for intrusion detection. In *International Workshop on Recent Advances in Intrusion Detection*, pages 265–283. Springer, 2009.

[295] Vicki S Chambers, Giovanni Marsico, Jonathan M Boutell, Marco Di Antonio, Geoffrey P Smith, and Shankar Balasubramanian. High-throughput sequencing of dna g-quadruplex structures in the human genome. *Nature biotechnology*, 33(8):877, 2015.

[296] Heidi Ledford. Personalized cancer vaccines show glimmers of success. `https://www.nature.com/news/personalized-cancer-vaccines-show-glimmers-of-success-1.22249`, 2017.

[297] Ross Overbeek, Niels Larsen, Gordon D Pusch, Mark D'Souza, Evgeni Selkov Jr, Nikos Kyrpides, Michael Fonstein, Natalia Maltsev, and Evgeni Selkov. Wit: integrated system for high-throughput genome sequence analysis and metabolic reconstruction. *Nucleic acids research*, 2000.

[298] Michela Becchi and Patrick Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the 2007 ACM CoNEXT conference*, page 1. ACM, 2007.

[299] Jan van Lunteren and Alexis Guanella. Hardware-accelerated regular expression matching at multiple tens of gb/s. In *Proceedings of IEEE INFOCOM*, pages 1737–1745. IEEE, 2012.

[300] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3088–3098, 2014.

[301] John L Hennessy and David A Patterson. A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development. In *Intl. Symp. on Comptr Architctre (ISCA'18)*, 2018.

[302] Reetinder Sidhu and Viktor K Prasanna. Fast regular expression matching using fpgas. In *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*. IEEE, 2001.

[303] Chengcheng Xu, Shuhui Chen, Jinshu Su, Siu-Ming Yiu, and Lucas CK Hui. A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms. *IEEE Communications Surveys & Tutorials*, 18(4):2991–3029, 2016.

[304] Chunkun Bo, Vinh Dang, Ted Xie, Jack Wadden, Mircea Stan, and Kevin Skadron. Automata processing in reconfigurable architectures: In-the-cloud deployment, cross-platform evaluation, and fast symbol-only reconfiguration. *ACM Transactions on Reconfigurable Technology and Systems*, 12(2):1–25, 2019.

[305] Lei Jiang, Qiong Dai, Qiu Tang, Jianlong Tan, and Binxing Fang. A fast regular expression matching engine for NIDS applying prediction scheme. In *Proceedings of the IEEE Symposium on Computers and Communications (ISCC)*, pages 1–7, 2014.

[306] Kubilay Atasu, Raphael Polig, Christoph Hagleitner, and Frederick R Reiss. Hardware-accelerated regular expression matching for high-throughput text analytics. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–7. IEEE, 2013.

[307] Jarrod Bakker, Bryan Ng, Winston KG Seah, and Adrian Pekar. Traffic classification with machine learning in a live network. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 488–493. IEEE, 2019.

[308] Marco Paolieri, Ivano Bonesana, and Marco Domenico Santambrogio. Recpu: A parallel and pipelined architecture for regular expression matching. In *Vlsi-Soc: Advanced Topics on Systems on a Chip*. Springer, 2009.

[309] Ivano Bonesana, Marco Paolieri, and Marco D Santambrogio. An adaptable fpga-based system for regular expression matching. In *2008 Design, Automation and Test in Europe*, pages 1262–1267. IEEE, 2008.

[310] Russ Cox. Regular expression matching: the virtual machine approach. `http://swtch.com/rsc/regexp/regexp2.html`, 2009.

[311] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14. IEEE, 2014.

[312] John E Hopcroft. *Introduction to automata theory, languages, and computation*. Pearson Education India, 2008.

[313] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, June 1968.

[314] Russ Cox. Regular expression matching with a trigram index or how google code search worked. `https://swtch.com/~rsc/regexp/regexp4.html`, 2012.

[315] Alex Roichman and Adar Weidman. Regular expression denial of service, 2012.

[316] Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu Feng, and Michela Becchi. Demystifying automata processing: Gpus, fpgas or micron's ap? In *Proceedings of the International Conference on Supercomputing*. ACM, 2017.

[317] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan: a fast multi-pattern regex matcher for modern cpus. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 631–648, 2019.

[318] Qiu Tang, Lei Jiang, Xin-xing Liu, and Qiong Dai. A real-time updatable fpga-based architecture for fast regular expression matching. *Procedia Computer Science*, 31:852–859, 2014.

[319] J. Yang, L. Jiang, Q. Tang, Q. Dai, and J. Tan. PiDFA: A practical multi-stride regular expression matching engine based on FPGA. In *Proceedings of the IEEE International Conference on Communications (ICC)*, pages 1–7, 2016.

[320] Evangelia A Sitaridi and Kenneth A Ross. Gpu-accelerated string matching for database applications. *The VLDB Journal*, 2016.

[321] Benjamin C Brodie, David E Taylor, and Ron K Cytron. A scalable architecture for high-throughput regular-expression pattern matching. *ACM SIGARCH computer architecture news*, 2006.

[322] Norio Yamagaki, Reetinder Sidhu, and Satoshi Kamiya. High-speed regular expression matching engine using multi-character nfa. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 131–136. IEEE, 2008.

[323] Ken Eguro. Automated dynamic reconfiguration for high-performance regular expression searching. In *Field-Programmable Technology, 2009. FPT 2009. International Conference on*. IEEE, 2009.

[324] Jens Teubner, Louis Woods, and Chongling Nie. Skeleton automata for fpgas: reconfiguring without reconstructing. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 229–240. ACM, 2012.

[325] Yusaku Kaneta, Shingo Yoshizawa, Shin-ichi Minato, Hiroki Arimura, and Yoshikazu Miyanaga. Dynamic reconfigurable bit-parallel architecture for large-scale regular expression matching. In *Field-Programmable Technology (FPT), 2010 International Conference on*. IEEE, 2010.

Bibliography

[326] Reza Rahimi, Elaheh Sadredini, Mircea Stan, and Kevin Skadron. Grapefruit: An open-source, full-stack, and customizable automata processing on fpgas. In *Proceedings of the IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 138–147, 2020.

[327] Paul Dlugosch et al. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3088–3098, 2014.

[328] Ke Wang, Kevin Angstadt, Chunkun Bo, Nathan Brunelle, Elaheh Sadredini, Tommy Tracy, Jack Wadden, Mircea Stan, and Kevin Skadron. An overview of micron's automata processor. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 1–3, 2016.

[329] Hiroki Nakahara, Tsutomu Sasao, and Munehiro Matsuura. A regular expression matching circuit based on a decomposed automaton. In *International Symposium on Applied Reconfigurable Computing*. Springer, 2011.

[330] Jan Van Lunteren, Christoph Hagleitner, Timothy Heil, Giora Biran, Uzi Shvadron, and Kubilay Atasu. Designing a programmable wire-speed regular-expression matching accelerator. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 461–472. IEEE Computer Society, 2012.

[331] Hsiang-Jen Tsai, Chien-Chih Chen, Yin-Chi Peng, Ya-Han Tsao, Yen-Ning Chiang, Wei-Cheng Zhao, Meng-Fan Chang, and Tien-Fu Chen. A flexible wildcard-pattern matching accelerator via simultaneous discrete finite automata. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(12):3302–3316, 2017.

[332] Kanak Agarwal and Raphael Polig. A high-speed and large-scale dictionary matching engine for information extraction systems. In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*. IEEE, 2013.

[333] Xuan-Thuan Nguyen, Hong-Thu Nguyen, Katsumi Inoue, Osamu Shimojo, and Cong-Kha Pham. Highly parallel bitmap-based regular expression matching for text analytics. In *Circuits and Systems (ISCAS), 2017 IEEE International Symposium on*, pages 1–4. IEEE, 2017.

[334] Google. Google re2. `https://github.com/google/re2`, 2020.

[335] Vern Paxson, W Estes, and J Millaway. Flex: the fast lexical analyzer. *URL http://flex. sourceforge. net*, 2012.

[336] Jeffrey Brown, Sandra Woodward, Brian Bass, and Charlie Johnson. Ibm power edge of network processor: A wire-speed system on a chip. *IEEE Micro*, 31(2):76–85, 2011.

[337] S Van Doren. Hoti 2019: Compute express link. In *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 18–18. IEEE Computer Society, 2019.

[338] National Center for Biotechnology Information. Homo sapiens chromosome dataset. `https://tinyurl.com/cromoOld`, 2017.

[339] Escherichia coli uniprotkb dataset. `https://www.uniprot.org/uniprot/?query=ecoli&sort=score`, 1996.

[340] Christian JA Sigrist, Edouard De Castro, Petra S Langendijk-Genevaux, Virginie Le Saux, Amos Bairoch, and Nicolas Hulo. Prorule: a new database containing functional and structural information on prosite profiles. *Bioinformatics*, 21(21):4060–4066, 2005.

[341] Jongse Park, Hardik Sharma, Divya Mahajan, Joon Kyung Kim, Preston Olds, and Hadi Esmaeilzadeh. Scale-out acceleration for machine learning. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 367–381, 2017.

[342] Michela Becchi and Patrick Crowley. Efficient regular expression evaluation: theory to practice. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 50–59. ACM, 2008.

[343] Donald E Knuth. On the translation of languages from left to right. *Information and control*, 8(6):607–639, 1965.

[344] M. Abbas and V. Betz. Latency insensitive design styles for fpgas. In *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 360–3607, 2018.

[345] Indranil Roy. *Algorithmic techniques for the micron automata processor*. PhD thesis, Georgia Institute of Technology, 2015.

[346] Apple. Apple M1. Small chip. Giant leap. `https://web.archive.org/web/20201110184757/https://www.apple.com/mac/m1/`, 2020. Last accessed 23 September 2021.

[347] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steve Reinhardt, Adrian Caulfield, Eric Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th International Symposium on Computer Architecture, 2018*. ACM, June 2018.

[348] Guozhu Xin, Jun Han, Tianyu Yin, Yuchao Zhou, Jianwei Yang, Xu Cheng, and Xiaoyang Zeng. Vpqc: A domain-specific vector processor for post-quantum cryptography based on risc-v architecture. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(8):2672–2684, 2020.

[349] Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. Anydsl: A partial evaluation framework for programming high-performance libraries. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–30, 2018.

[350] Jian Fang, Yvo TB Mulder, Jan Hidders, Jinho Lee, and H Peter Hofstee. In-memory database acceleration on fpgas: a survey. *The VLDB Journal*, 29(1):33–59, 2020.

[351] Dong Uk Lee, Kyung Whan Kim, Kwan Weon Kim, Kang Seol Lee, Sang Jin Byeon, Jae Hwan Kim, Jin Hee Cho, Jaejin Lee, and Jun Hyun Chun. A 1.2 v 8 gb 8-channel 128 gb/s high-bandwidth memory (hbm) stacked dram with effective i/o test circuits. *IEEE Journal of Solid-State Circuits*, 50(1):191–203, 2014.

[352] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaaauw, and Reetuparna Das. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 383–396. IEEE, 2018.

[353] Nastaran Hajinazar, Geraldo F Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. Simdram: a framework for bit-serial simd processing using dram. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–345, 2021.

[354] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. A modern primer on processing in memory. *arXiv preprint arXiv:2012.03112*, 2020.

[355] Gagandeep Singh, Mohammed Alser, Damla Senol Cali, Dionysios Diamantopoulos, Juan Gómez-Luna, Henk Corporaal, and Onur Mutlu. Fpga-based near-memory acceleration of modern data-intensive applications. *IEEE Micro*, 2021.

Bibliography

[356] ARM. ARM big.LITTLE: Processing Architectures for Power Efficiencies and Performance. https://www.arm.com/why-arm/technologies/big-little, 2012. Last accessed 5 October 2021.

[357] 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.

[358] Burkhard Ringlein, Francois Abel, Alexander Ditter, Beat Weiss, Christoph Hagleitner, and Dietmar Fey. System architecture for network-attached fpgas in the cloud using partial reconfiguration. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 293–300. IEEE, 2019.

[359] MSR. Honeycomb. https://www.microsoft.com/en-us/research/project/honeycomb/, 2020.

[360] Dionysios Diamantopoulos, Raphael Polig, Burkhard Ringlein, Mitra Purandare, Beat Weiss, Christoph Hagleitner, Mark Lantz, and Francois Abel. Acceleration-as-a-{\mu} service: A cloud-native monte-carlo option pricing engine on cpus, gpus and disaggregated fpgas. *arXiv preprint arXiv:2106.06293*, 2021.

[361] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[362] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. The risc-v instruction set manual, volume i: Base user-level isa. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 116, 2011.

[363] Krste Asanović and David A Patterson. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[364] Xifan Tang, Edouard Giacomin, Baudouin Chauviere, Aurelien Alacchi, and Pierre-Emmanuel Gaillardon. Openfpga: An open-source framework for agile prototyping customizable fpgas. *IEEE Micro*, 40(4):41–48, 2020.

[365] Ang Li and David Wentzlaff. Prga: An open-source framework for building and using custom fpgas. In *The First Workshop on Open-Source Design Automation; Florence, Italy*, pages 1–6, 2019.

[366] Xilinx. Vitis HLS Front-end is Now Open Source. https://github.com/Xilinx/HLS, 2021. last accessed: 2 July 2021.

[367] QuickLogic. QuickLogic Open Reconfigurable Computing (QORC) MCU + eFPGA SoC Open Source Software Tools. https://www.quicklogic.com/software/qorc-mcu-efpga-fpga-open-source-tools/. last accessed: 2 July 2021.