UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

Lina: a fast design optimisation tool for software-based FPGA programming

André Bannwart Perina

Tese de Doutorado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

ICMC USP SÃO CARLOS

André Bannwart Perina

Lina: a fast design optimisation tool for software-based FPGA programming

Thesis submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP – in accordance with the requirements of the Computer and Mathematical Sciences Graduate Program, for the degree of Doctor in Science. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Vanderlei Bonato

USP – São Carlos
August 2022

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

P445l

Perina, André Bannwart
Lina: a fast design optimisation tool for software-based FPGA programming / André Bannwart Perina; orientador Vanderlei Bonato. -- São Carlos, 2022.
189 p.

Tese (Doutorado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2022.

1. FPGA. 2. High-level synthesis. 3. Design space exploration. 4. Synthesis-less design exploration. I. Bonato, Vanderlei, orient. II. Título.

Bibliotecários responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2:
Gláucia Maria Saia Cristianini - CRB - 8/4938
Juliana de Souza Moraes - CRB - 8/6176

André Bannwart Perina

Lina: uma ferramenta de otimização de projeto para programação de FPGAs baseada em software

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Vanderlei Bonato

USP – São Carlos
Agosto de 2022

ACKNOWLEDGEMENTS

The path that resulted in this thesis includes several contributions in various forms. Here is the acknowledgment for those.

First, I would like to thank Prof. Vanderlei Bonato for the significant insight, suggestions and guidelines given through these years. A great thank you also goes to Prof. Jürgen Becker for the great advisorship given through the last years. Also Prof. Eduardo Marques, Prof. Alexandre Delbem, Prof. João Cardoso and Prof. Pedro Diniz for all the research lessons that I gathered in my academic years.

I also appreciate all the support, both research and emotional, of all the people I've met in the research community. A special thank you to the Laboratório de Computação Reconfigurável (LCR), including Carlos, Leandro, Erinaldo, Marcilyanne. Also my great appreciation for the Institute for Information Processing Technologies of the Karlsruhe Institute of Technology (ITIV-KIT), that provided significant support for this work (thank you Arthur, Augusto, Birgitta, Fabian, Florian, Jens, Kevin, Steffen, and all of ITIV).

I also leave my deep thank you to all family and friends who have been involved in this journey. To my family in São Vicente / Santos and Indaiatuba, and specially to my brother, my mother and my father for always being around when needed. Thank you to the friends in São Carlos for all the experience (thank you Gaperia for all friendship!), to the friends in Indaiatuba, São Vicente / Santos, Karlsruhe. You all have been vital through this journey. A special thank you to my friends Carol, Filipe and Maria for all the patience, friendship and support.

In addition, I would like to thank all the infrastructural and financial support given for this research:

- To the institutes and universities ICMC-USP and ITIV-KIT;
- To the "Fundação de Amparo à Pesquisa do Estado de São Paulo" (FAPESP) for the significant support given through processes no. 2016/18937-7 and 2018/22289-6;
- To the "Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil" (CAPES), which partially financed this project through finance code 001;
- To the "Conselho Nacional de Desenvolvimento Científico e Tecnológico" (CNPq) that financed the early steps of my graduate program;

- To the “Paderborn Center for Parallel Computing” and “Intel Labs Academic Compute Environment”, for providing access to the Intel Hardware Accelerator Research Program (HARP) resources.

At last, I thank the examination board for the insights and suggestions provided during defence.

"All for freedom and for pleasure
Nothing ever lasts forever
Everybody wants to rule the world"
(Tears for Fears)

RESUMO

PERINA, A. B. **Lina: uma ferramenta de otimização de projeto para programação de FPGAs baseada em software**. 2022. 189 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2022.

A contínua jornada da indústria de semicondutores levou ao desenvolvimento de diversas arquiteturas alternativas para uma computação eficiente. "Field-Programmable Gate Arrays" (FPGAs) e "Graphics Processing Units" (GPUs) são exemplos de dispositivos utilizados para acelerar aplicações. FPGAs são capazes de oferecer um paralelismo massivo para tarefas adequadas quando apropriadamente programados. No entanto, projetar para FPGA não é trivial e requer um conhecimento específico que foge do desenvolvimento usual em software. Como uma alternativa buscando aumentar a programabilidade, ferramentas de Síntese de Alto Nível (do inglês "High-Level Synthesis", ou HLS) permitem o uso de linguagens de alto-nível como C/C++/OpenCL para programar FPGAs. No entanto, experimentos preliminares e outros estudos na literatura demonstram que ainda são necessárias diversas modificações no código de alto nível para que os resultados sejam minimamente aceitáveis. Tal aspecto mitiga a democratização e simplificação propostas pelas ferramentas HLS. A contribuição principal desta tese considera C/C++ como linguagem de entrada HLS, e é composta por uma ferramenta de exploração de espaço de projeto acoplada à um estimador denominado Lina. Baseado no estimador Lin-analyzer, Lina usa a execução instrumentada de um código em alto-nível para aproximar o método de compilação do Vivado HLS, um compilador HLS C/C++ para FPGAs da Xilinx. Para um dado kernel C/C++, Lina calcula uma rápida estimativa para métricas de tempo de execução e recursos de FPGA ocupados. Junto com diretivas de otimização usadas pelo compilador HLS que o Lina também suporta, a metodologia aqui proposta permite a otimização não apenas do tempo de execução, mas também de recursos lógicos de FPGA. Considerando 16 kernels C/C++ do benchmark PolyBench, as soluções estimadas como ótimas pelo Lina estiveram dentro de 1% das melhores opções consideradas. Uma média de $14-16\times$ de speedup de performance foi atingida, o que representa 70% do valor máximo alcançável considerando os espaços de projeto explorados. Adicionalmente, Lina suporta a exploração de transações com memórias off-chip em busca de otimizações como coalescência, empacotamento de dados, ou até informar sobre potenciais limitações do compilador HLS que possam degradar a performance.

Palavras-chave: FPGA, Síntese de alto nível, Exploração de espaço de projeto, Exploração sem síntese de projeto.

ABSTRACT

PERINA, A. B. **Lina: a fast design optimisation tool for software-based FPGA programming**. 2022. 189 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2022.

The continuous technology push on the semiconductor industry has led to the development of several alternate architectures for efficient computing. Field-Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs) are examples of devices used to accelerate applications. FPGAs are able to provide massive parallelism for suitable tasks when properly programmed. However, designing for FPGA is non-trivial and requires specific knowledge that deviates from the usual software programming. As an alternative towards increasing programmability, High-Level Synthesis (HLS) tools allow high-level languages such as C/C++/OpenCL to be used as input for FPGA design. However, early experiments and other studies in the literature demonstrate that significant code modification is still necessary so that the results are minimally acceptable. This aspect mitigates the democratisation and simplification that HLS tools seek to achieve. The major contribution of this thesis works on the C/C++ level, composed of a design space exploration tool that uses an estimator named Lina. Based on Lin-analyzer, Lina uses a traced execution of a software code to approximate the compilation behaviour of Vivado HLS, a C/C++ HLS compiler for Xilinx FPGAs. For a given C/C++ kernel, Lina provides a fast approximation of metrics such as execution time and FPGA resources occupied. Along with HLS compiler optimisation directives that Lina supports in its estimation, our exploration method allows the optimisation of not only execution time, but also FPGA resource usage. We then used Lina to optimise 16 C/C++ kernels from the PolyBench benchmark, and the estimated optimal solutions were among the 1% best options. An average of $14-16\times$ performance speedup was achieved, accounting for 70% of the reachable speedup when considering the traversed design spaces. Additionally, Lina allows the exploration of off-chip memory transactions in search of optimisations such as coalescing, data packing, or to inform about potential HLS compiler limitations that could degrade performance.

Keywords: FPGA, High-level synthesis, Design space exploration, Synthesis-less design exploration.

LIST OF FIGURES

LIST OF CHARTS

LIST OF ALGORITHMS

LIST OF SOURCE CODES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ADRS	Average Distance from Reference Set
ALAP	As-Late-As-Possible
ANN	Artificial Neural Network
AOCL	Altera (Intel FPGA) SDK for OpenCL
ASAP	As-Soon-As-Possible
BRAM	Block RAM
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DDDG	Dynamic Data Dependency Graph
DFG	Data-Flow Graph
DSP	Digital Signal Processor
DVFS	Dynamic Voltage Frequency Scaling
FF	Flip-Flop
FN	Fast Newman
FPGA	Field-Programmable Gate Array
FU	Functional Unit
GPU	Graphics Processing Unit
HBM	High-Bandwidth Memory
HLS	High-Level Synthesis
II	Initiation Interval
IR	Intermediate Representation
ITRS	International Technology Roadmap for Semiconductors
KUT	Kernel Under Test
LUT	LookUp Table
LVQ	Learning Vector Quantisation
MLP	Multi-Layer Perceptron
NCD	Normalised Compression Distance
NJ	Neighbour Joining
NOD	Near-Optimal Density
NVML	NVIDIA Management Library
RBF	Radial Basis Function

RCLS	Resource-Constrained List Scheduling
RTL	Register-Transfer Level
SDC	System of Difference Constraints
SIMD	Single-Instruction, Multiple Data
SoC	System-on-Chip
TDP	Thermal Design Power

CONTENTS

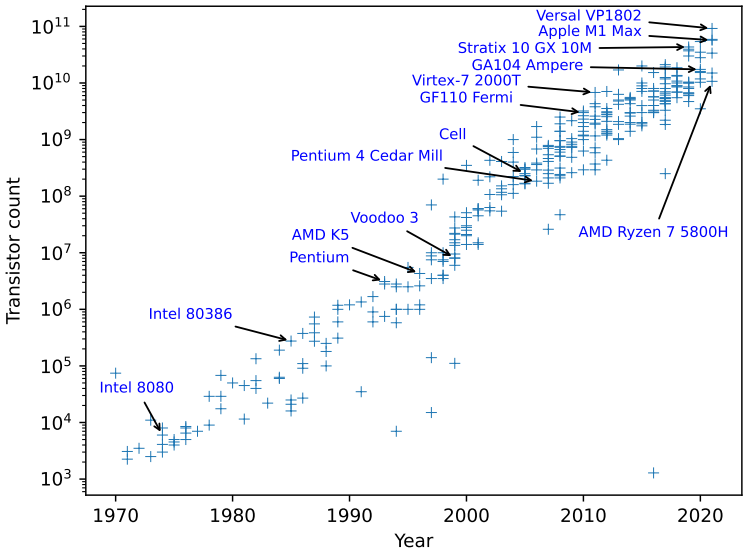
CHAPTER

1

INTRODUCTION

Digital electronic devices are omnipresent in most of modern human society. From staircase light timers to military-grade equipment, the capability and complexity of these devices greatly evolved during the last decades. This evolution is intimately related to the basic building block of every digital circuit: the transistor. Improvements in the fabrication process allowed smaller transistors to be produced, which in turn allowed the production of more complex and faster circuits. Figure 1 shows how the transistor count evolved during the years for common processing units of computers, phones, etc. The transistor count increased from few thousands in the 1970s to billions in late 2010s: a 7× order of magnitude increase in the span of 50 years.

Figure 1 – Transistor count of several integrated circuits over the last decades.



Source: Adapted from Roser and Ritchie (2013), Wikipedia (2022).

Still in the 1970s, two "laws" were stated by famous researchers that provided a good insight on what was about to happen in the following years. The first one is the well-known Moore's law, which states that the number of transistors in a dense integrated circuit would double at roughly every two years (MOORE, 1965). The second law is the Dennard (or MOSFET) scaling, which roughly states that as transistors get smaller, their power density stays constant (DENNARD *et al.*, 1974). This has a twofold effect:

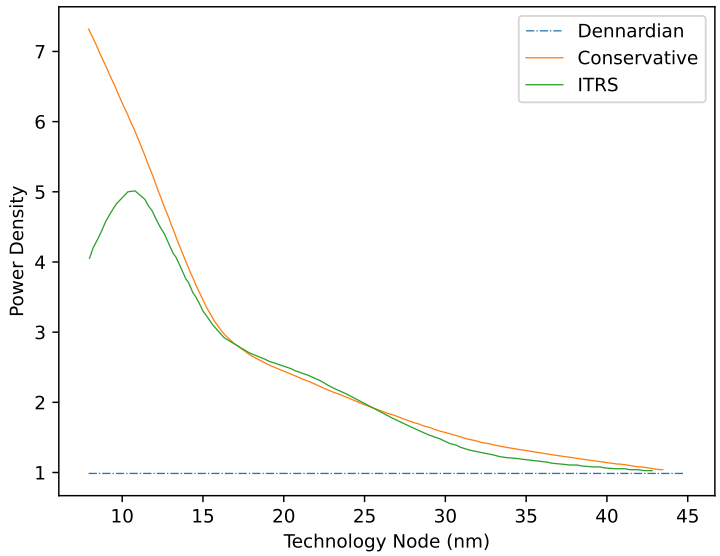
- The reduction of the transistor's size causes a reduction on the circuit's delay (since the paths are shorter), which in turn allows higher operating frequencies for the circuit. The reduced size also contributes to a reduced capacitance;
- The reduction of operating voltage, coupled with the reduced capacitance, contributes to lower energy consumption.

For roughly 30 years, the industry constantly reduced transistor sizes, increased transistor count on a microchip and increased the operating frequency. This allowed a constant evolution on processing capability while the energy consumption was still manageable. As the transistor sizes started to reach a few dozen nanometres, issues started to appear and reduce this downscaling trend. The leakage current of the transistors started to become more pronounced and coupled with the high operating frequencies, the energy efficiency of new devices started to worsen. This trend can be clearly seen in Figure 2, where the power density over different technology sizes is presented (KANDURI *et al.*, 2017). It is quite visible that the power density — as monitored by the International Technology Roadmap for Semiconductors (ITRS) — was not kept constant as proposed by the Dennard scaling. This increase caused issues not only on supplying proper power the whole circuit, but also issues on heat dissipation.

Therefore, the industry started to take different paths in order to circumvent the power density increase. Industry and academic efforts were directed towards finding technologies that could also improve the energy efficiency of the computations performed. Mainstream computer processors — the Central Processing Unit (CPU) — started to include features such as multi-core processing, Dynamic Voltage Frequency Scaling (DVFS) among other technologies that allowed to improve the energy efficiency.

Interest on other architectures also grew. Graphics Processing Units (GPU), for example, became increasingly ubiquituous for parallel computing applications. The GPU's architecture is highly suitable for matrix/vector operations due to its Single-Instruction, Multiple-Data (SIMD) nature. Before the 2000s, GPUs were solely used for graphical purposes of showing user interfaces, or as graphics accelerators for games and visual computation. Larsen and McAllister (2001) presented the first attempt to use a GPU as a computation platform. In this early experiment, a matrix-matrix multiply was performed

Figure 2 – Power density evolution over different transistor sizes. ITRS estimations are from 2013 (Semiconductor Industry Association, 2013) and conservative estimations are from Borkar (2010).



Source: Adapted from Kanduri *et al.* (2017).

using a GPU. Since the GPU up to this point was solely used for graphical purposes, several workarounds were done in order to adapt the inputs and outputs (for example, the input matrices must first be converted to image-like data structures that are readable by the GPU's graphical pipeline). Nonetheless, performance speedup is expected as compared to a matrix multiply performed in a CPU. During the following years, the GPU hardware evolved and programming languages were developed in order to provide a friendlier interface to the GPU as a matrix calculator. Examples of these languages include CUDA and OpenCL.

1.1 Field-Programmable Gate Arrays and High-Level Synthesis

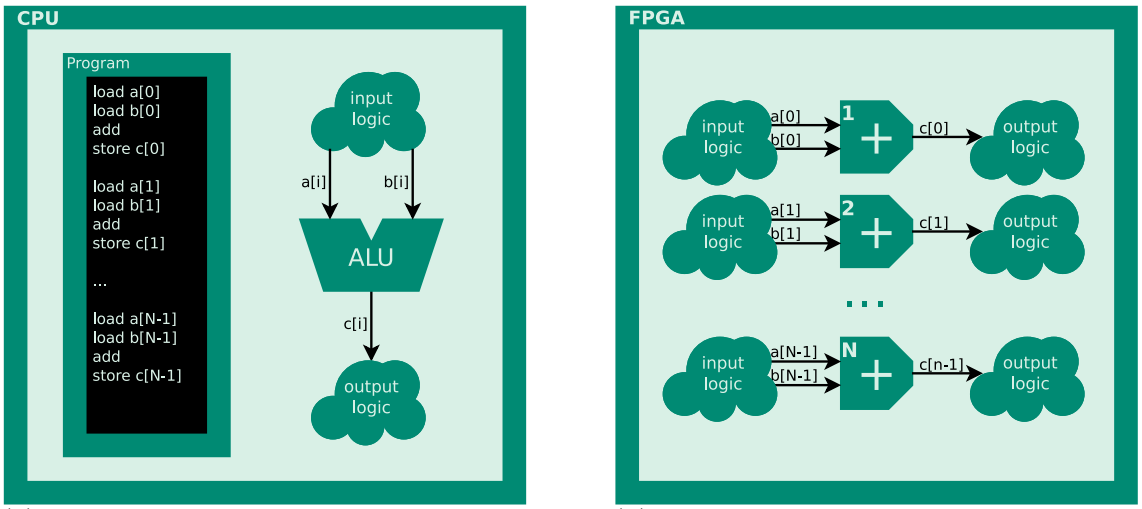
The Field-Programmable Gate Array (FPGA) is a device able to implement logic circuits and can be used in several applications ranging from arithmetic computations, glue logic, network-on-chips, etc. The FPGA is composed of an array of small components such as LookUp Tables (LUT), Flip-Flops (FF), Digital Signal Processors (DSP), and Block RAMs (BRAM). Additionally, a complex on-chip network system is provided to allow the routing between these components.

The FPGA technology is not recent, however it received more attention in the last years due to its natural low-power aspect. FPGAs operate in a frequency range that

peaks around 500 MHz (HUTTON, 2015), and therefore the heat dissipation that is generated proportional to the frequency is greatly reduced, as compared to CPUs and GPUs that operate in the GHz level. FPGAs use the same transistor techonology as from CPUs/GPUs and therefore they also suffer from static power leakage, which has been increasing over the last years (SEIFOORI *et al.*, 2018).

FPGAs differ significantly from CPUs in terms of programming. While on a CPU the developer writes a program containing a sequence of instructions to perform a given task, in the FPGA the developer must define computation units, the wiring between them, storage logic (registers, BRAMs) and the input/output interface. Considering a vector-add example for arrays with size N, Figure 3 shows the implementation approach on CPU and FPGA. For the FPGA, one must define add units, insert intermediate registers (if applicable), define the inputs/outputs, and connect all components.

Figure 3 – Simple depiction of a vector add example on CPU and FPGA.



(a) Example on CPU: the vector add is described as a sequential list of instructions that operates an Arithmetic Logic Unit (ALU).

(b) Example on FPGA: the vector add is performed in parallel, N adders are instantiated.

Source: Elaborated by the author.

Modern CPUs are equipped with optimisations that do allow some parallelism and/or acceleration from sequential software (e.g. multiple cores, out-of-order execution). On the other hand, FPGAs enable true parallelism with an extended degree of freedom, as long as the device's resource and routing budget allows. For the example presented in the last figure, the FPGA design performs all adds in parallel.

FPGA designs are usually described using Register-Transfer Level (RTL) languages. These languages allow the developer to define the computation units, wires, registers, multiplexers, etc. RTL programming differs significantly from software. The developer must be aware of several constraints that are not common in the software world (e.g. timing constraints, synchronisation between different parallel parts of the design, etc.).

Furthermore, debugging is tedious and time-consuming due to the significant time spent on compilation[1].

The input/output interfacing is also not trivial. The designs programmed on an FPGA have direct access to input/output pins of the chip, which must then be interfaced accordingly (e.g. connected to a system bus, or even to simple push switches and LEDs). This also adds another layer of complexity for debugging, since internal values are usually not accessible without probes. However, there are modern FPGAs coupled with standard interfaces that provide a more intuitive usage. For example the Xilinx Zynq UltraScale+ family of System-on-Chips (SoC) includes an ARM processor and an FPGA. Using vendor-specific drivers, it is possible to execute a Linux operating system on this platform, and use Linux-based APIs from Xilinx to interface with the FPGA.

1.1.1 High-Level Synthesis

FPGAs have been for a long time a niche architecture, used mostly by hardware design experts. Due to their non-trivial programming using RTL, they are still rather unpopular among high-level developers. As an alternative, High-Level Synthesis (HLS) tools provide means of transforming high-level input codes into FPGA hardware designs. Many HLS tools have been released over the last decades. Examples of popular HLS tools include Intel FPGA SDK for OpenCL, Xilinx Vivado HLS[2], and LegUp.

Given a high-level source code, the HLS compiler applies several analyses to identify data dependencies between operations. Then, non-dependent operations are scheduled to occur in parallel and are allocated to synthesisable computation units known as Functional Units (FU). The customisable aspect of FPGAs allows the HLS compiler to freely decide the amount and type of FUs while balancing resource usage, routing complexity and timing constraints. Figure 4 presents a simplified view of a design generated by HLS compilers. A Finite State Machine (FSM) created by the compiler coordinates which data should be retrieved, computed, and stored at each clock cycle.

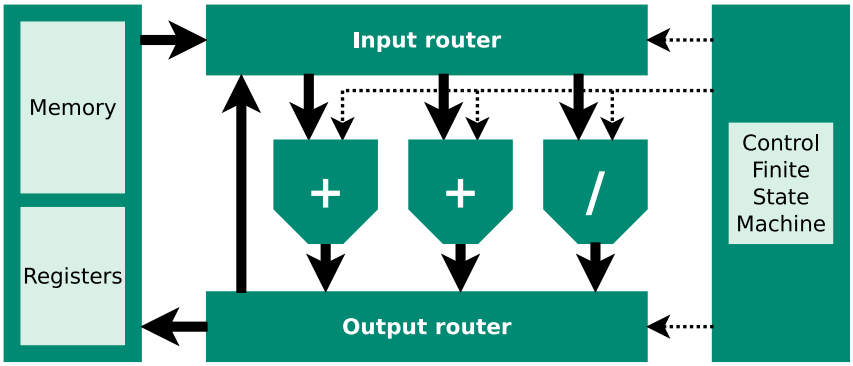
The output of the HLS compiler is an automatically generated hardware description (usually as an RTL code). Then, the automatically-generated RTL code is fed to the usual FPGA design synthesis that maps all RTL components to physical FPGA resources. Although HLS is usually not as time consuming as the whole FPGA synthesis and place-route process, it can still take significant time depending on the complexity of the input code (ZHONG *et al.*, 2016).

Performance is highly dependent on how well the HLS compiler extracts parallelism from the code and how well it adapts to the FPGA constraints. Software codes

[1] Being a resource allocation and routing problem, FPGA synthesis may take from few minutes to several hours, and might even fail due to failed constraints.

[2] Recently renamed to Xilinx Vitis HLS.

Figure 4 – Basic depiction of the hardware generated by an HLS compiler. Continuous lines represent data lanes, and dotted lines represent control lanes. Three FUs are shown in this example: two adders and one divider.



Source: Perina *et al.* (2021).

frequently contain structures and patterns that are dynamically defined (e.g. a dynamic data dependency, dynamic memory allocation). In these cases, the HLS compiler will either fail or take conservative paths that may severely impact the final performance. Hardware-orientated optimisations — either through manual code rewrite or through the use of compiler directives — are essential to achieve reasonable performance (ZOHOURI *et al.*, 2016; MUSLIM *et al.*, 2017; WELLER *et al.*, 2017).

1.2 Motivation and Objective

Multiple studies (ZOHOURI *et al.*, 2016; MUSLIM *et al.*, 2017; WELLER *et al.*, 2017) have shown that code optimisations for hardware generation are essential to achieve reasonable results when using HLS. Early experiments for this thesis - as presented in Appendix C - further corroborate to this statement. This clearly puts a barrier on the FPGA democratisation that HLS tools seek to achieve. FPGA accessibility is increased by generating hardware from software code, but decreased by requiring hardware-specific optimisations that are exotic to the software world.

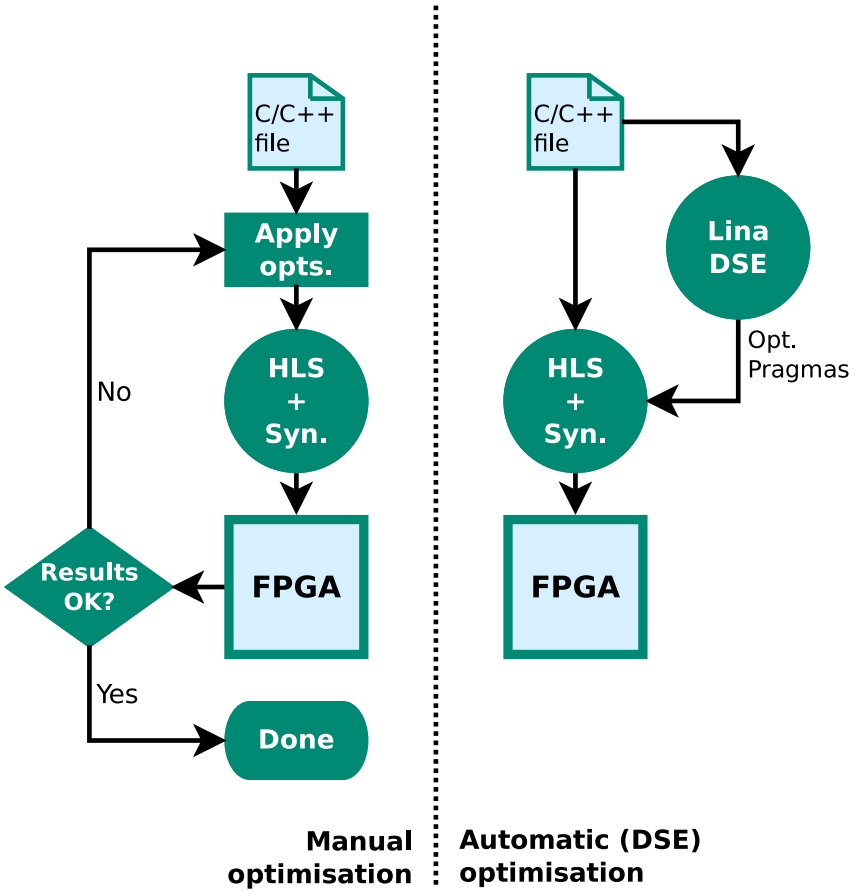
This thesis seeks the goal of further reducing the hardware burden on the software developer during HLS design. **The main objective is to provide a fast optimisation approach to automatically improve the quality of FPGA designs generated from HLS tools.**

To achieve this goal, a Design Space Exploration (DSE) framework using an estimator named Lina was developed (PERINA *et al.*, 2021). Lina is a fast estimator forked from Lin-analyzer (ZHONG *et al.*, 2016) with several improvements. Lina is considered fast since it does not require design synthesis or HLS compilation for its estimations, which in turn allows the exploration of large design spaces in feasible time. Our DSE framework targets Xilinx Vivado HLS, a popular HLS compiler that supports C/C++/OpenCL as

input. The design space consists of toggling compiler directives such as loop pipeline, circuit operating frequency, etc. These directives are unnatural to the software developer, and our approach reduces this burden by automatically performing a benefit analysis for each combination of directives.

Figure 5 presents a comparison between the normal development flow with HLS and the proposed approach using Lina. In the normal development flow, the developer must manually apply the compiler directives and evaluate the outcome. Since the combination of compiler directives can easily grow in size, the developer must perform this step repeatedly until a suitable combination is found. When using the proposed approach, this exploration is performed automatically and the best combination is supplied at end, with no need of manual iterative search. Additionally, it can traverse large design spaces orders of magnitude faster than the manual exploration, since Lina does not rely on design synthesis to evaluate each combination.

Figure 5 – Comparison between manual optimisation approach (left) and the automatic optimisation approach using the proposed DSE framework (right).



Source: Elaborated by the author.

1.3 Thesis Structure

In Chapter 2, we present the literature review for this thesis, including the HLS compilers that we studied for our approach and optimisation frameworks on the HLS level. Our initial approach was to use a machine learning framework for an OpenCL HLS compiler. However, we noticed that performance in FPGA was severely limited and significant code optimisation would be required. Our findings were further confirmed by a performance study on several OpenCL kernels found in the literature. This first machine learning framework is presented in Appendix B, and the performance study is presented in Appendix C.

The performance study has shown that existing OpenCL kernels that do not originally target FPGA might require significant optimisations in order to have reasonable performance. Our approach then shifted towards considering C/C++ kernels (or functions) as input and a different HLS compiler. This resulted in the DSE framework with Lina, and it is presented in Chapter 3. Finally, we perform a final discussion and conclude the thesis in Chapter 4.

Appendix A presents the published contributions and a list of all tools developed. Appendix D presents supplemental material to the Lina content. In Appendix E, we compare the performance of applications optimised for FPGA using our approach against the same applications optimised for GPU.

CHAPTER

2

LITERATURE REVIEW

In this chapter, we present relevant work related to this thesis. First we present a selection of modern HLS compilers. Then, we present benchmarks and applications tested on HLS. Finally, we present tools to assist the development and optimise HLS designs.

2.1 High-Level Synthesis Compilers

Development of high-level synthesis tools span for almost three decades. Early HLS compilers include Celoxica's Handel-C and Impulse C. Most of these tools are either defunct or currently not maintained (NANE *et al.*, 2015).

The Xilinx Vitis HLS (also known by its former name Xilinx Vivado HLS) is one of the most popular commercial HLS frameworks still active. Its base software was acquired by Xilinx from AutoESL in 2011, when the tool was still named Autopilot (Xilinx, Inc., 2011). Vivado HLS accepts C/C++ as input, and is integrated to several larger frameworks from Xilinx such as SDx and Versal. This contributes to its popularity, since an out-of-box integration is provided with Xilinx FPGAs.

Alternatively, OpenCL is supported as language input for Xilinx SDx and Intel FPGA (former Altera) SDK for OpenCL (AOCL). This enables an automatic support to a plethora of existing OpenCL kernels, mostly developed for GPU architectures. Furthermore, the OpenCL API provides an integrated solution for compiling and programming FPGAs using OpenCL. For example, the FPGA board can be connected via PCI Express to a host machine. This host machine is responsible for executing a software code that manages and dispatches OpenCL kernels to the FPGA board. The AOCL API performs these tasks seamlessly, requiring little effort from the user for defining the communication interface from/to the FPGA.

LegUp (CANIS *et al.*, 2011) is an example of academic C/C++ HLS tool that

gained popularity in the previous decade. Its open-source nature allowed for third-party researches and improvements (ROSA; BOUGANIS; BONATO, 2018; RAMANATHAN; CONSTANTINIDES; WICKERSON, 2018). However, LegUp closed its source in 2017 (CHOI, 2017) and was acquired by Microchip in 2020.

2.2 High-Level Synthesis Applications and Comparison Studies

Several studies demonstrate the applicability of HLS in modern applications, for example k-nearest neighbour (PU *et al.*, 2015), stereo correspondence matching (TATSUMI *et al.*, 2015), sparse matrix multiplication (GIEFERS *et al.*, 2016), and tsunami simulation (KONO *et al.*, 2018). These studies all include two common flavours: manual HLS-orientated optimisations; and that FPGA may lose in performance against other accelerators, but may win in energy efficiency.

High-level synthesis based on OpenCL has the advantage of enabling an automatic support to a variety of OpenCL kernels already available, mostly for GPU platforms. Examples of OpenCL benchmarks include Rodinia (CHE *et al.*, 2009) and the Scalable Heterogeneous Computing Benchmark Suite (SHOC) (DANALIS *et al.*, 2010). Rodinia is composed of 23 applications with implementations in OpenMP, CUDA and OpenCL. SHOC is a wider benchmark that may scale to more than one accelerator using MPI and also includes stability tests.

Similarly, benchmarks such as PolyBench (POUCHET, 2012) or Parboil (STRATTON *et al.*, 2012) can be used with C/C++ HLS compilers. The PolyBench is a C/C++ floating-point application suite from various domains (linear algebra, image processing, etc.) with static control flows and affine loop bounds. The Parboil benchmark is a suite of applications that are presented with different variants targetting different architectures. Parboil may be used to compare accelerators by executing the right version for each (e.g. CUDA for NVIDIA GPUs or OpenMP for multi-core CPUs), and it also includes the baseline sequential C/C++ versions.

Although HLS allows the portability of high-level codes to FPGAs, the performance is not automatically portable. The Rodinia and SHOC kernels, for example, are designed in a SIMD fashion that greatly favours GPUs. Zohouri (2018) ported six of the Rodinia applications to FPGA by applying aggressive code transformations. Results show that for five applications, peak FPGA performance was only achieved through complete code redesign. Similarly, Muslim *et al.* (2017) compared test cases from three application classes on FPGA and GPU, using the optimised code for each. Weller *et al.* (2017) used a partial differential equation solver as comparison study between CPU, GPU and FPGA. Two different HLS compilers and FPGAs were used, with their findings showing that HLS

performance is not even portable among different vendors. The authors point that if the FPGAs had DDR4 or faster memories, they would be more competitive against modern GPUs that include very fast memories.

2.3 High-Level Synthesis Assist Tools

From the work presented in the previous section and considering our findings that we present in Appendix C, it is clear that optimisation efforts — either manual or automatic — are essential to achieve reasonable results with HLS. In this section, we present related work that is part of the HLS optimisation field. First, we present HLS estimators that can be used to approximate useful metrics such as execution time or consumed energy. Second, we present HLS optimisation frameworks.

2.3.1 High-Level Synthesis Estimators

Estimators are of great interest for the HLS community. They apply simpler, faster models than the ones used by HLS compilers, while still approximating useful metrics with reasonable accuracy. The estimations can be used for early design insights, potentially reducing project design iterations and time-to-market. Additionally, they can be used to guide DSE processes.

We separate the work here in two classes: static and dynamic approaches. Static approaches use only the information provided in the source code. Dynamic approaches use traced software executions to guide the estimation.

Considering static approaches, Enzler *et al.* (2000) present a model to predict several hardware metrics from Data-Flow Graphs (DFG). Quantitative features are extracted from the DFG and fed to a set of equations that provide a rough estimation for area, latency and achievable frequency. Similarly, Kulkarni *et al.* (2006) present a compile-time area estimation for an HLS framework named Cameron. The DFG is generated as a part of the compilation process, which is then evaluated using regression models in order to provide a LUT estimation. Bilavarn *et al.* (2006) present an estimator for area/delay tradeoffs that uses a hierarchical control-data dependency graph, which is generated by performing a depth-first search on a C code.

Related to the dynamic approaches, Bjureus, Millberg and Jantsch (2002) present a framework to schedule trace-based DFGs from Matlab specifications. Their model is similar to Enzler *et al.* (2000) as both do not consider memory resources and control logic overheads. Shao *et al.* (2014) present Aladdin, an estimator that uses C traces to predict area, performance and power on the electronic design level. Lin-Analyzer (ZHONG *et al.*, 2016) is similar to Aladdin, however it targets FPGAs instead of chip design level, and it includes optimisation directives that are FPGA-orientated.

2.3.2 High-Level Synthesis Optimisation Frameworks

Estimator results are useful during HLS development, however they still require interpretation possibly tied to hardware expertise. Alternatively, estimators can be attached to optimisation frameworks that automatically control the estimator inputs and/or digest the results. In turn, these frameworks are able to provide a clearer optimisation path to the developer. In this section we present tools that perform optimisations on HLS applications. Most of the work presented perform optimisation via DSE, that is, these tools traverse a design space composed of HLS optimisation knobs in search for an optimal combination. Estimators (or even the early HLS compiling reports) are used to evaluate different design points and to take decisions.

A DSE specific to Convolutional Neural Networks (CNN) is presented by Zhang *et al.* (2015). It uses a static approach coupled with the roofline model to explore loop ordering, tiling size, on-chip buffering, loop unroll and pipeline. Their optimised solution for the AlexNet CNN has a throughput of 61.62 Giga-Operations per Second (GOPS), outperforming previous FPGA implementations while having 4.8× speedup and 24.6× less energy consumption than its software counterpart.

MPSeeker (ZHONG *et al.*, 2017) is presented as a multi-level parallelism explorer. It uses Lin-analyzer as its performance and DSP/BRAM estimator and a machine learning approach to estimate FFs/LUTs. Synthesis is not used during the DSE phase (though required for training the model once), enabling an exploration of 280 design points under few minutes for some kernels tested. MPSeeker explores designs composed of several processing elements and uses resource estimation to keep the solutions under feasible levels.

A lattice-traversing approach is proposed by Ferretti, Ansaloni and Pozzi (2018), with the key difference of not being coupled to any accelerator or specific HLS tool. The design space is mapped to a lattice representation and a traversal methodology is presented, on which sample points of the design space are characterised in order to orientate the navigation. Experimental results with 5 C kernels from the CHStone benchmark show that their results are close to the true optimal in terms of execution time and LUT usage. At least 50 design points were synthesised for each kernel.

Choi and Cong (2018) present a DSE with support for variable loop bounds. They use Vivado HLS's software simulation flow to profile and extract loop information, such as runtime trip count. Then selected points of the design space are compiled with HLS, which are used through analytical models to characterise the whole design space. They also propose source-to-source transformations based on predefined code patterns to further improve performance. For five PolyBench kernels, an average speedup of 75× was achieved compared to the baseline HLS version. The exploration time ranged from 5 to 20 minutes, with design spaces containing around 100 points (and one kernel with up to 1000 points).

COMBA (ZHAO *et al.*, 2019) is an analytical design space explorer targetting Vivado HLS that supports a variety of compiler pragmas. Their DSE methodology is metric-guided, discarding design points that are marked as uninteresting. No synthesis is required, which makes COMBA explore hundreds of points in just a few seconds. COMBA's optimisation problem focuses on minimising the design's latency, while considering the total LUT, DSP and BRAM count of a target platform as constraints. According to experimental results, COMBA reaches speedups above 100× for several PolyBench kernels.

FlexCL (LIANG; WANG; ZHANG, 2018) is presented as an analytical performance and power model for OpenCL HLS. Their exploration is focused on the computation grid by changing parameters including work-group size, work-item and work-group pipeline, compute unit parallelism, etc. Evaluated using the Rodinia benchmark, their model present an average percentage error of 9.5% and 12.6% for performance and power, respectively, and can explore design of spaces of 128 points within 3 and 6 minutes.

Oppermann *et al.* (2019) present SkyCastle, a resource-aware multi-loop HLS scheduler. SkyCastle tackles loop pipelining using an Integer Linear Programming method to pipeline multiple loop nests. Their findings show that SkyCastle is able to generate comparable results to the Vivado HLS standard scheduler, however using less resources. Furthermore, SkyCastle is able to pipeline complex loop nests that Vivado HLS is not able to schedule due to resource constraints violation.

Boyi (JIANG *et al.*, 2020) is an OpenCL HLS optimiser that explores and searches for predefined computation patterns. Boyi then decides the most suitable OpenCL execution model for this kernel between SIMD and single work-item, while also attempting to utilise FPGA-specific optimisations (such as channels) to improve performance.

2.4 Final Remarks

Since AOCL provides an almost automatic support to existing OpenCL kernels in FPGA, our first approach was to provide a framework able to perform an early decision of which accelerator would fit best for a given OpenCL kernel. However, as shown in the previous sections, OpenCL kernels without any specific optimisation tend to perform poorly on FPGAs. Our intention then was to develop and attach an optimisation model to the AOCL framework, and our early decision framework would take such optimisation in consideration. However, our findings in Appendix C have shown that significant manipulations are required to transform existing OpenCL kernels — often modelled for GPUs — to a format more suitable for FPGAs. We decided to take a better approach and use a high-level language that better fits HLS, such as sequential C/C++. This led to the development of a design space exploration framework using our estimator Lina, as presented in Chapter 3.

CHAPTER

3

FAST DESIGN SPACE OPTIMISATION FOR C/C++ HLS USING LINA

As presented in the previous chapter, high-level source codes targetting FPGAs via HLS require a significant amount of hardware-orientated optimisations to achieve reasonable results. An automatic optimisation model is desirable in this case to improve the quality of HLS-generated designs. In this chapter, we present our DSE approach that considers C/C++ as HLS language input. The design space is created by varying compiler directives (loop unroll, loop pipeline, array partition) and the circuit's target operating frequency. The core of our approach is Lina, an estimator that allows exploring the design space orders of magnitude faster than synthesising and evaluating each design point (combination of directives). This approach of avoiding C/C++ synthesis during DSE has a great potential towards reducing the current time gap between software and hardware development. Our timing model supports different clock frequencies, and the resource model supports both floating-point and integer datapaths. Additionally, Lina is able to evaluate off-chip memory accesses, estimate coalescing optimisations and provide reports with warnings about failed memory optimisations.

Lina is based on Lin-analyzer (ZHONG *et al.*, 2016): it inherits the trace-based scheduling while adding features such as support to non-perfect loop nests, different operating frequencies and a lightweight model for resource estimation of the most common elements of modern FPGAs: LUTs, FFs, DSPs and BRAMs. The memory model leverages the information generated through trace to evaluate potential memory optimisations or bottlenecks. In addition, common data structures between design point estimations are reused to further reduce the exploration time.

We evaluate the Lina framework through three studies. In the first study (PERINA; BECKER; BONATO, 2019a), we perform a small design space exploration (i.e. 48 points) to test the timing and non-perfect loop models, in comparison to the original Lin-analyzer

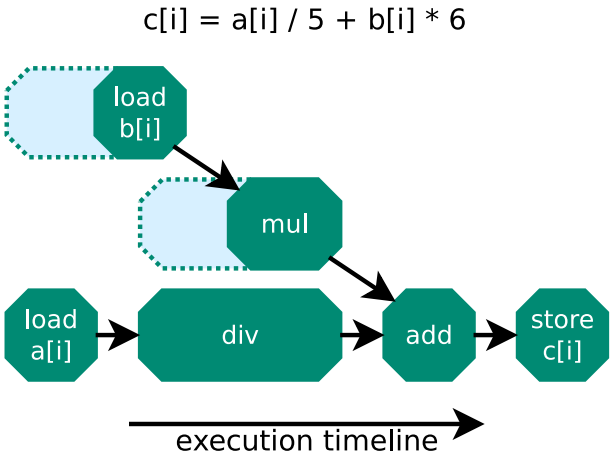
that does not include these features. In the second study, we focus on the usage of Lina for design space exploration (PERINA *et al.*, 2021) considering hundreds of design points. We extend the optimisation objectives to reduce not only the design execution time[1], but also the resource footprint. In both previous studies, it is considered that all input/output arrays are located in on-chip FPGA memory blocks. As a final study, we consider off-chip memory accesses and use Lina to explore potential coalescing and data packing optimisations.

The next sections are structured as follows: section 3.1 presents a brief background of the HLS functionality and how Lin-analyzer approximates the HLS behaviour. Then, section 3.2 describes Lina and the proposed DSE method, section 3.3 presents the experimental setup, followed by the validation results in section 3.4. Finally, section 3.5 presents our final considerations, including a comparison of our approach against related work.

3.1 Overview of High-Level Synthesis and Estimation

The HLS compilation process can be roughly divided in two steps: scheduling and binding. During scheduling, the compiler identifies the instructions in a software code; finds dependencies between them; and generates a draft schedule for the identified instructions. Figure 6 presents a schedule example for one arithmetic statement. The instructions are laid down forming the execution timeline that the HLS compiler generates. Some of the instructions have a "schedule window", that is, they can be executed at a sooner or later point without interfering with their dependencies.

Figure 6 – Example of a simple function and its dependency graph. The dashed region of the nodes represent the schedule window and the edges represent the data dependencies.



Source: Elaborated by the author.

[1] To avoid confusion, we refer to "design execution time" as being the execution time of the generated HLS designs, and "exploration execution time" to the time spent by our DSE to explore the design space.

Instruction scheduling is performed using similar tactics as for task scheduling problems. One approach is to use As-Soon-As-Possible (ASAP) and As-Late-As-Possible (ALAP) algorithms. These allow to find the soonest and the latest point that every instruction can execute while still respecting the dependencies. Therefore, ASAP/ALAP outputs the schedule window for every instruction. Another approach is to use System of Difference Constraints (SDC). In this case, the scheduling constraints are mapped to a series of inequalities $x_i - x_j \leq b_{ij}$ (x_i, x_j are free parameters and b_{ij} is a constraint), and a solver is used to find a feasible solution.

Up to this point, the scheduling only takes into account the data and control dependencies extracted from the code. In the binding step of HLS, the physical constraints of FPGAs are taken into account. For example, FPGA BRAM modules are usually dual-ported, meaning that no more than two separate accesses are allowed in a same clock cycle. The unconstrained scheduling from the previous step must then be adjusted to the FPGA constraints. One approach is to use Resource-Constrained List Scheduling (RCLS), where an unconstrained scheduling is iteratively traversed, and instructions that violates FPGA constraints are delayed until the restrictions are lifted. If using SDC, additional inequations related to the physical FPGA constraints can be added to the system.

Moreover, the binding step is responsible for mapping each instruction to a physical Functional Unit (FU) which executes that instruction on FPGA. FUs are often reused by different instructions to reduce resource idling and routing complexity. For each instruction in the schedule, the HLS compiler checks whether there is a free FU to execute that instruction or not. If yes, then the instruction is mapped to that FU. If not, the HLS compiler evaluates whether it is better to instantiate a new FU, or to delay the execution of that instruction until another busy FU becomes available.

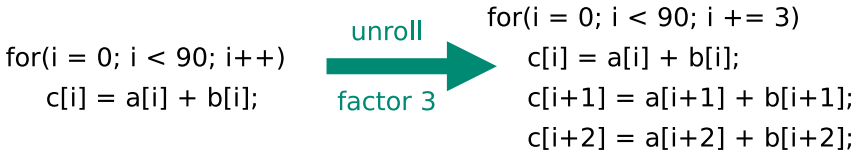
In addition to optimising an application for HLS through code rewrite, the HLS compilers usually provide a set of compiler directives — or pragmas[2] — that can be used to automatically boost the parallelism extraction. The three most common are loop unroll, loop pipeline and array partition.

By default, the HLS compiler generates the compute module for a single iteration of a loop and repeatedly executes it for all iterations. When a loop is unrolled, the loop body is replicated by a factor n. The HLS compiler then attempts to schedule and bind n iterations at once, which can improve performance at the cost of increased resource usage. Figure 7 presents an example of loop unroll with factor 3.

In a non-pipelined loop, the i-th iteration only starts when the $i-1$-th iteration finishes. With loop pipelining, the HLS compiler attempts to generate a schedule that overlaps the execution of successive loop iterations. The pipeline's efficiency is dictated

[2] The expressions "compiler directive" and "pragma" are interchangeably used in this thesis.

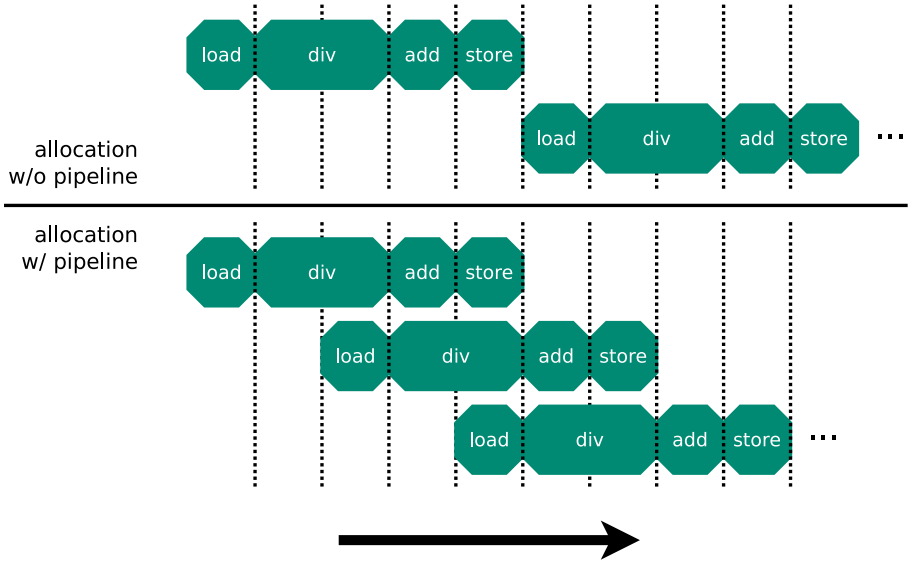
Figure 7 – Example of loop unroll with factor 3.



Source: Elaborated by the author.

by the number of clock cycles required to start a new iteration, also known as Initiation Interval (II). An ideal pipeline design has an II of 1, i.e. a new loop iteration is started at every clock cycle. Figure 8 exemplifies the concept of loop pipeline.

Figure 8 – Example of loop scheduling without and with pipeline directive (new iteration starting every 2 cycles).

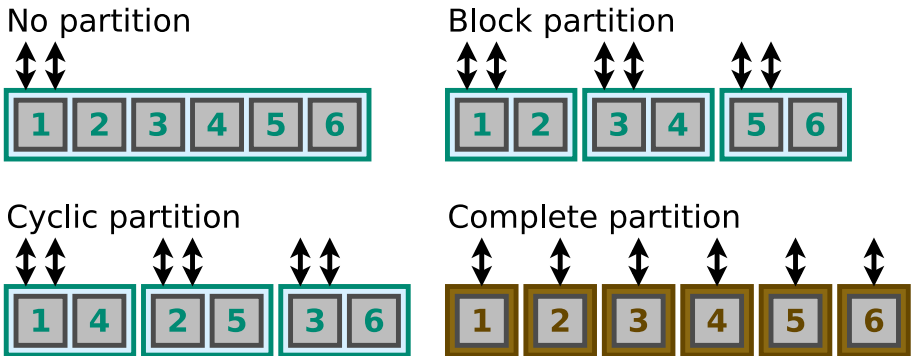


Source: Elaborated by the author.

Although unroll and pipeline directives can increase performance, they can be bottlenecked if there is data contention (that is, if the design is not able to deliver the inputs at the required rate, and/or write the results as soon as they're computed). As previously mentioned, BRAM memory modules have a physical limited amount of read-/write ports. By using array partition, additional BRAM modules can be used to hold one array, increasing the amount of ports available per clock cycle. Depending on the memory access pattern, the increased amount of ports can alleviate memory contention. Figure 9 presents an example of how an array can be partitioned to different configurations. Which configuration suits best depends on how the data is accessed by the code.

The design space formed by exploring the aforementioned compiler directives is

Figure 9 – Example of different partitioning configurations for an array. The arrows represent the available read/write ports per BRAM block.



Source: Elaborated by the author.

well-suited for design space exploration, since the space can easily explode in size. This would incur in unfeasible exploration times if every design point were to be evaluated using HLS synthesis, thus motivating the use of design space exploration heuristics and fast HLS estimators. In the following section, we provide a brief overview of the Lin-analyzer estimator and how it approximates its behaviour to the actual HLS compilation process of Vivado HLS.

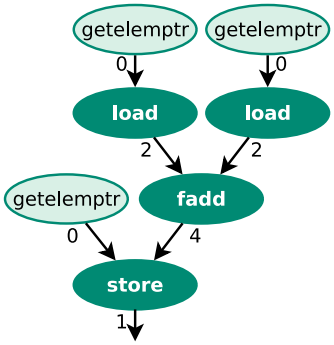
3.1.1 Lin-analyzer Overview

High-Level Synthesis compilation requires complex steps for generating functionally correct designs. However, simpler models can skip these steps while still approximating metrics with reasonable accuracy. Some may use dependency graphs for estimation, which can be generated either from static code analysis (ENZLER *et al.*, 2000; KULKARNI *et al.*, 2006; BILAVARN *et al.*, 2006) or from software traces (BJUREUS; MILLBERG; JANTSCH, 2002; SHAO *et al.*, 2014; ZHONG *et al.*, 2016). One advantage of the latter is that global code motion optimisation is inherently enabled, as all control and false data dependencies are implicitly resolved by the execution that generates the software trace. The resultant graph (henceforth called Dynamic Data Dependency Graph, or DDDG) provides an optimistic notion of the parallelism capabilities of the code, which can then be constrained to reflect realistic parallel architectures (AUSTIN; SOHI, 1992).

Lin-analyzer (ZHONG *et al.*, 2016) is a trace-based estimator that approximates the latency of HLS-generated designs from C codes. First, an input C/C++ function is converted to the LLVM's Intermediate Representation (IR) language using the LLVM's CLANG frontend compiler. Then, Lin-analyzer injects trace functions in the IR code, executes it, and generates a dynamic trace that contains all executed instructions and the data they accessed. Lin-analyzer then uses the dynamic trace to infer data dependencies and generate a DDDG, where each node represents an executed LLVM IR instruction.

For example, Figure 10 shows a DDDG generated from a vector add.

Figure 10 – Example of DDDG generated from the LLVM IR of a vector add.



(a) DDDG for one loop iteration. The weights represent the latency of FUs.

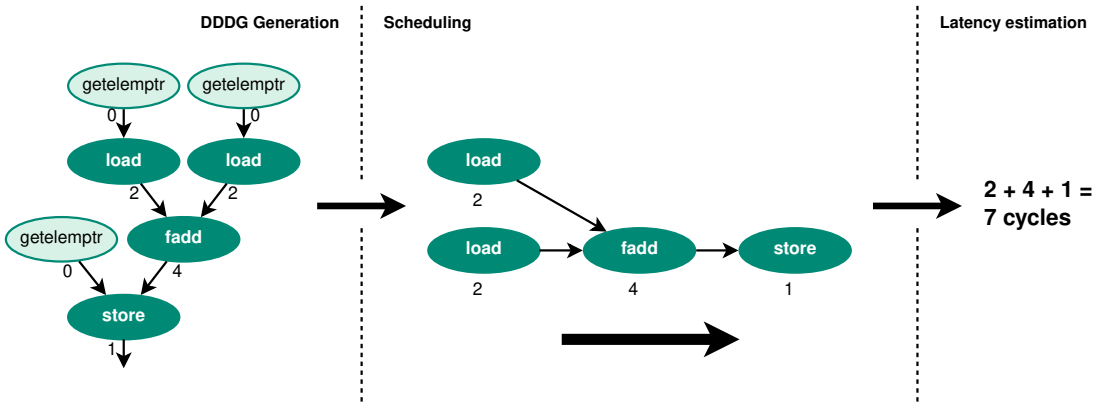
```
; for(int i = 0; i < 1024; i++)
;     c[i] = a[i] + b[i];
body:
  %ai = getelementptr float, float* %a, i64 %i
  %av = load float, float* %ai, align 4
  %bi = getelementptr float, float* %b, i64 %i
  %bv = load float, float* %bi, align 4
  %cv = fadd float %av, %bv
  %ci = getelementptr float, float* %c, i64 %i
  store float %cv, float* %ci, align 4
  br i1 %footer
```

(b) C/C++ snippet of the vector add (first two lines) and the respective LLVM IR code for the loop body (below).

Source: Perina *et al.* (2021).

Then, Lin-analyzer optimises the DDDG by removing redundant loads/stores and nodes that have no relation to hardware generation (e.g. software debug nodes). The remaining nodes are annotated with the latency required to solve each instruction according to a target platform. The latencies used by Lin-analyzer are based on the FUs used by Xilinx Vivado HLS at an operating frequency of 100MHz. Then, Lin-analyzer applies the ASAP, ALAP and RCLS scheduling algorithms to constrain the DDDG according to the constraints of the target platform. Finally, the latency can be derived from the constrained DDDG graph, which approximates the cycle count of the actual design that Vivado HLS would generate for the same input code. Figure 11 presents a simplified example of a DDDG being scheduled and the latency calculation.

Figure 11 – Example of a DDDG scheduling and its latency calculation. In this example, the two `load` nodes are for different arrays and thus allowed to occur in parallel. Nodes with no relation to hardware generation (e.g. the `getelementptr` instructions) are removed prior to scheduling).



Source: Elaborated by the author.

The key aspect of Lin-analyzer is that it also supports the compiler directives previously mentioned. They are implemented as follows:

- **Loop unroll:** by default, Lin-analyzer only generates the DDDG for a single iteration of the innermost loop body. This is similar to the scheduling performed by Vivado HLS when no unroll is enabled. If unroll is enabled by a factor of n, Lin-analyzer reads n iterations from the dynamic trace and generate a single DDDG schedule for them. The resultant scheduling is similar to the one performed by Vivado HLS when an unroll of n is enabled.

- **Loop pipeline:** after DDDG scheduling, Lin-analyzer calculates the pipelined total latency l_{total} as follows:

$$l_{total} = II * (TC - 1) + l_{iter} \tag{3.1}$$

  where II is the initiation interval, TC is the loop's trip count and l_{iter} is the latency estimated from the DDDG for a single loop iteration.

  The value of II can be found by effectively performing the pipeline scheduling of the DDDG. However this is a complex task, and it can be avoided by approximating II to a best-case scenario minimum initiation interval MII. Lin-analyzer calculates MII based on two factors that often limit the reachable minimum value of II: recurrence and resource constraints.

  The recurrence-constrained MII ($RecMII$) considers the case where a dependency across loop iterations limits the MII value (i.e. if a loop iteration depends on data from previous iterations, it cannot start before these values are calculated). Lin-analyzer calculates $RecMII$ by generating an additional DDDG scheduling that includes the double of iterations than the original DDDG. Then, $RecMII$ is calculated by subtracting the scheduled latency of both DDDGs. If there is no dependency between iterations, both DDDGs should have similar cycle count due to independent parallelism, and $RecMII$ tends to zero. Conversely, a large latency difference between the DDDGs is an indication that there could be dependency between loop iterations and thus $RecMII$ is adjusted accordingly. Both DDDGs here considered are unconstrained (i.e. before RCLS).

  The second value that influences MII is the resource-constrained MII ($ResMII$). This constraint is based on physical resource limitations on the FPGAs: memory ports ($ResMII_{mem}$) and FU budget ($ResMII_{fu}$). When pipeline is enabled, execution of multiple loop iterations overlap, which in turn increases resource occupancy. As an example, consider that a DDDG has 4 different reads for a same array, and that the array is stored in a BRAM module that allows up to two simultaneous reads per clock cycle. It is impossible to schedule a pipeline with II lower than 2, since when

the pipeline is completely filled, a scheduling with $II = 1$ would require four BRAM simultaneous reads. The same logic applies for FU occupancy.

Finally, MII is calculated by considering the worst-case constraint from the ones mentioned above, as follows (ZHONG *et al.*, 2016; LI *et al.*, 2015; RAU, 1994):

$$MII = \max\{RecMII, ResMII\} \tag{3.2a}$$

$$ResMII = \max\{ResMII_{mem}, ResMII_{fu}\} \tag{3.2b}$$

$$ResMII_{mem} = \max_{m}\left\{\left\lceil\frac{Nr_m}{Pr_m}\right\rceil, \left\lceil\frac{Nw_m}{Pw_m}\right\rceil\right\} \tag{3.2c}$$

$$ResMII_{fu} = \max_{q}\left\{\left\lceil\frac{FUu_q}{FUc_q}\right\rceil\right\} \tag{3.2d}$$

where: Nr_m and Pr_m are the number of reads and number of read ports for array m; Nw_m and Pw_m are the respective counterparts for write transactions; FUu_q (FU unlimited) is the largest amount of simultaneous instructions of type q in the DDDG (calculated through ASAP scheduling); and FUc_q (FU constrained) is the actual amount of FUs of type q instantiated (according to RCLS scheduling).

- **Array partition:** the partitioning of an array affects the port availability depending on which values are accessed. A complete partitioning, for example, maps every array element to separate registers. Since the memory accesses are explicitly resolved in the dynamic trace, Lin-analyzer easily identifies which partition is being accessed for an array based on its resolved address, and maintains a separate port budget for every partition.

These pragmas only affect the DDDG scheduling, which means that the same dynamic trace can be reused to estimate the latency of different combinations. Since trace generation is the most time-consuming step and is executed only once, Lin-analyzer is able to quickly traverse through large design spaces composed by the combinations of optimisation pragmas.

3.2 DSE Methodology with Lina

Lina inherits Lin-analyzer's dynamic trace execution and DDDG optimisation/scheduling logic, while new features are implemented to provide further explorations possibilities. The main differences between Lina and Lin-analyzer are presented in Chart 1.

Our DSE workflow is presented in Figure 12. Given a C kernel as input, Lina first generates the trace file through profiled execution. Then for every combination of

Chart 1 – Differences between Lin-analyzer and Lina.

Lin-analyzer	**Lina**
It is assumed that the circuit will operate at 100MHz	The timing-constrained scheduler accepts a continuous range of clock frequencies and supports operation chaining
DDDG is generated only for the innermost loop body	The non-perfect loop analyser constructs the DDDGs for all code segments in a loop nest
Resource estimation considers floating-point FUs and on-chip arrays (only BRAM)	The resource awareness considers integer FUs, floating-point FUs, auxiliary logic and additional resources related to on-chip arrays
All arrays are considered to be on-chip	Off-chip arrays are supported, and the memory model evaluates the accesses to off-chip memory, identifying potential bottlenecks and optimisation opportunities

Source: Adapted from Perina *et al.* (2021).

pragmas supplied by a job dispatcher, Lina uses the trace to generate and schedule a set of DDDGs. These are used to derive resource count and latency estimates.

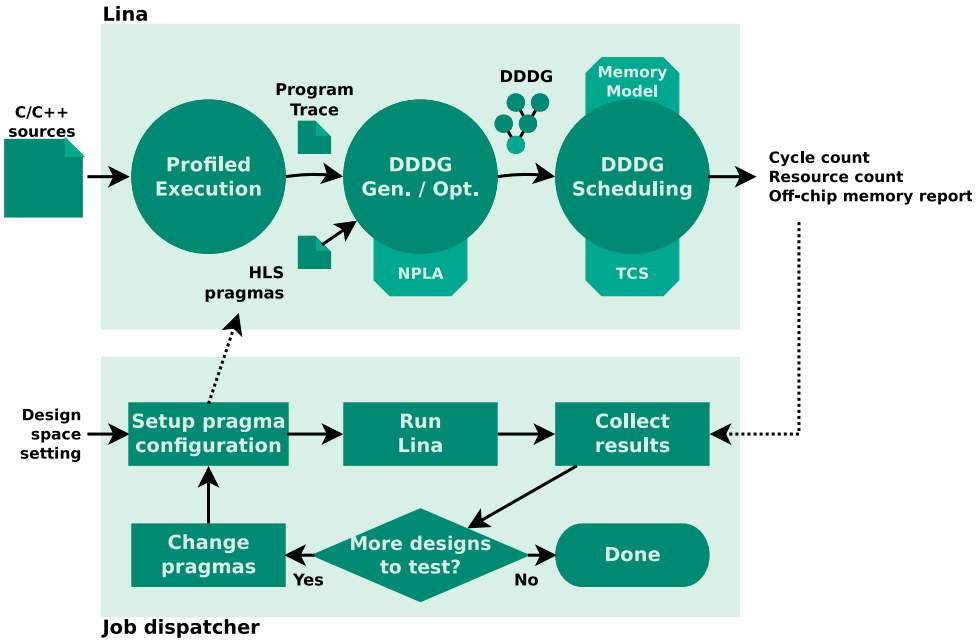
The design space is generated by toggling a series of predefined knobs: which loops should be pipelined and unrolled (up to a certain factor); which arrays should be partitioned (up to a certain factor and type of partition); and the operating frequency range. The job dispatcher exhaustively traverses this field of exploration by generating the combination of pragmas supplied to Lina. Since the estimation of each design point is independent, the job dispatcher performs a parallel exploration through p separate threads.

Cases with mutual pragma invalidation are discarded and not explored, those being:

- If pipeline is active for a certain loop level, all its subloops are automatically unrolled. Thus all design points that apply unroll pragmas to automatically unrolled loops are considered redundant and excluded;

- When a loop level l is fully unrolled, its logic is replicated for every iteration and the loop structure itself ceases to exist. Therefore, we discard the points where pipeline is active for a fully unrolled level, since there will be no actual loop to implement pipeline in this case.

Figure 12 – Lina flow: the trace is generated once for a given software code, then successive combinations of HLS pragmas are provided by the job dispatcher to estimate each design's latency and resource count. NPLA and TCS stand for Non-Perfect Loop Analyser and Timing-Constrained Scheduler, respectively.



Source: Adapted from Perina *et al.* (2021).

After all the valid design points are estimated, we define a Pareto set considering five objectives to minimise: design execution time, LUTs, FFs, DSPs and BRAMs.

In the following sections, we detail the key components of Lina: the timing-constrained scheduler, the non-perfect loop analyser, how the resource awareness model works, the off-chip memory model, and the cache mechanism implemented to reduce the DSE exploration time.

3.2.1 Timing-Constrained Scheduler

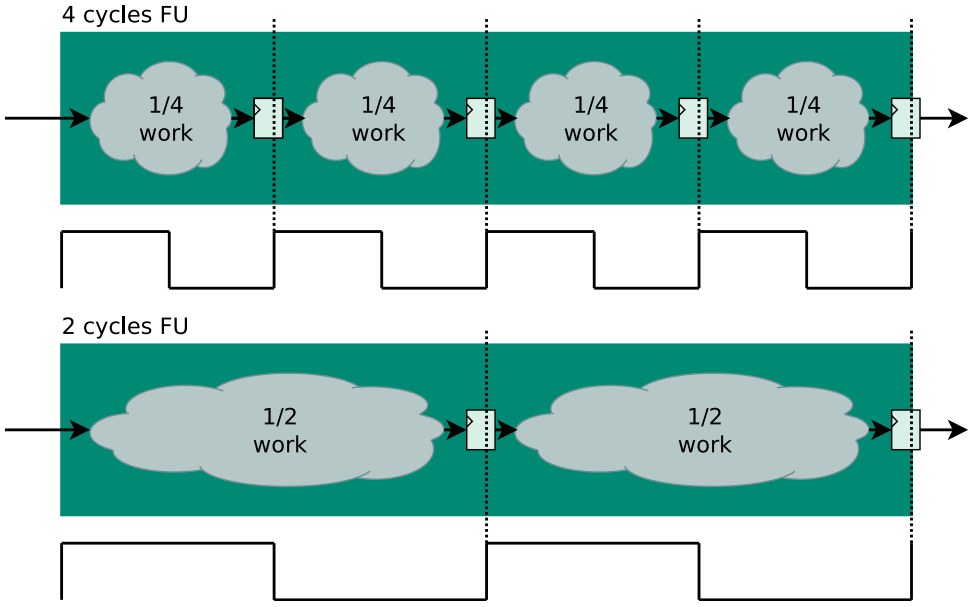
A higher clock frequency does not directly imply better performance, since the HLS compiler adapts the scheduling to comply with tighter constraints. The timing-aware model of Lina attempts to mimic this behaviour, which allows a more realistic evaluation under different timing constraints.

Timing-wise, a functional unit is defined by latency l and critical path delay t_{cp}. Latency defines the amount of clock cycles needed for the FU to generate a valid result from a given input, while t_{cp} defines how much time inside a clock cycle the FU takes to perform all required intermediate calculations. If this FU is operated with an operational frequency of f_{op} (and its respective period $t_{op} = f_{op}^{-1}$) so that $t_{cp} > t_{op}$, the FU is not able to finish all its internal operations before the start of the next clock cycle. The unfinished intermediate values are then propagated through the circuit, which causes

undefined behaviour.

Thus, the HLS compiler must guarantee that every FU is configured to perform its work within the timing budget of the target clock period (i.e. $t_{cp} < t_{op}$). Several HLS tools (including Vivado HLS) provide a set of configurations for each type of FU with different configurations of l and t_{cp}. A low value of l means that more work must be done per clock cycle (i.e. higher t_{cp}) and vice-versa. Figure 13 presents a graphical depiction on how l and t_{cp} interact when adjusted.

Figure 13 – Depiction of two configurations for one FU, with latencies of 4 (top) and 2 (bottom) cycles, respectively. Smaller latencies lead to greater work per cycle, which in turn constrains the frequency.



Source: Elaborated by the author.

The following subsections present the key aspects of the Timing-Constrained Scheduler, namely the hardware profile library, initial timing analysis and operation chaining.

3.2.1.1 Hardware Profile Library and FU Characterisation

Lina contains a hardware profile library that supports a continuous interval of frequencies ranging from 16.66MHz to 500MHz. For each FU supported by Lina, the library contains a set of latency-delay pairs $\{(l, t_{cp})\}$ that define the available configurations for the supported frequency interval.

To identify these configurations, we implement a method that iterates over different values of target frequencies and collects the timing information from the Vivado HLS reports. This is performed for every FU that Lina supports, and must be performed only when a new platform support is added. For example, Source code 1 presents a simple C

function used to instantiate an FU of type `fdiv`. This code is used for microbenchmarking, on which the Vivado report is parsed.

Source code 1 – Example of simple function used to characterise FUs

```
1: #define OP /
2: #define TYPE_T float
3:
4: void foo(TYPE_T A[1024], TYPE_T B[1024], TYPE_T C[1024]) {
5:   for(int i = 0; i < 1024; i++)
6:     C[i] = A[i] OP B[i];
7: }
8:
```

Vivado HLS emits a detailed report after the high-level synthesis compilation. This report includes the latencies and critical path delays of all FUs used in the code. These delays are not final, since they might still vary after design synthesis, place/route, and timing analysis. However these are the values considered by Vivado HLS to perform its scheduling and binding, which is what Lina actually approximates.

Our method performs a binary search in order to avoid an excessive amount of HLS runs, while still providing a complete characterisation. Algorithm 1 presents the recursive procedure used, where FU is the functional unit being analysed (e.g. `add`, `sub`), (l_{min}, t_{min}) is the FU configuration at the lower bound of the recursive search, (l_{max}, t_{max}) is the higher bound configuration, and C is the final set containing all identified configurations. The `HLSRUN(FU, t)` function represents a call to Vivado HLS considering an operating frequency of t^{-1}.

The algorithm works as follows. First, the FU is characterised on the bounds of the supported interval of Lina (i.e. $t_{lmin} = 500\text{MHz} = 2\text{ns}$ and $t_{lmax} = 16.66\text{MHz} = 60\text{ns}$)[3]. This provides the initial search bounds $(l_{min}, t_{min}) = (l_{lmin}, 2)$ and $(l_{max}, t_{max}) = (l_{lmax}, 60)$, where l_{lmin} and l_{lmax} are the latencies found through Vivado HLS at the bound periods. The recursive algorithm then executes Vivado HLS for the interval's midpoint $t_{mid} = t_{min} + \frac{t_{max} - t_{min}}{2}$. Depending on the configuration retrieved, the algorithm proceeds the search on two new intervals $[t_{min}, t_{mid}[$ and $]t_{mid}, t_{max}]$. Every new configuration found is appended to C during the search. Each recursion branch ceases itself if one of the conditions are met:

- If the interval being searched falls under a break condition size (i.e. $t_{max} - t_{min} < \varepsilon$,

[3] Note that we consider a period interval for the recursive search, not frequency. This means that the higher bound of the interval is actually the lower frequency, whereas the lower bound is the higher frequency.

line 4);

- If the midpoint configuration was previously included in C (line 10);
- If the midpoint latency found is one unit lower than l_{min} or one unit higher than l_{max} (i.e. if $l_{mid} = l_{min} - 1$ or $l_{mid} = l_{max} + 1$, lines 16 and 21)[4].

Algorithm 1 – The recursive exploration algorithm

```
1:  procedure EXPLORE(FU, (l_min,t_min), (l_max,t_max), C)
2:      t_sub ← t_max − t_min
3:      if t_sub < ε then
4:          return                    ▷ If interval is lower than ε, stop recursion
5:      end if
6:
7:      t_mid ← t_min + t_sub/2                              ▷ Calculate midpoint
8:      (l,t_cp) ← HLSRUN(FU, t_mid)                         ▷ Get HLS report for midpoint
9:      if (l,t_cp) ∈ C then
10:         return                    ▷ Repeated, no need to explore further
11:     else
12:         APPEND(C, (l,t_cp))                              ▷ Append new configuration found
13:     end if
14:
15:     if l_min = l + 1 then
16:         return          ▷ No need to explore between two consecutive integers
17:     else
18:         EXPLORE(FU, (l_min,t_min), (l,t_cp), C)          ▷ Continue recursion (lower)
19:     end if
20:     if l_max = l − 1 then
21:         return          ▷ No need to explore between two consecutive integers
22:     else
23:         EXPLORE(FU, (l,t_cp), (l_max,t_max), C)          ▷ Continue recursion (upper)
24:     end if
25: end procedure
```

Algorithm 2 presents the complete FU characterisation algorithm, including the first HLS runs and the first recursion call.

3.2.1.2 Initial Timing Analysis

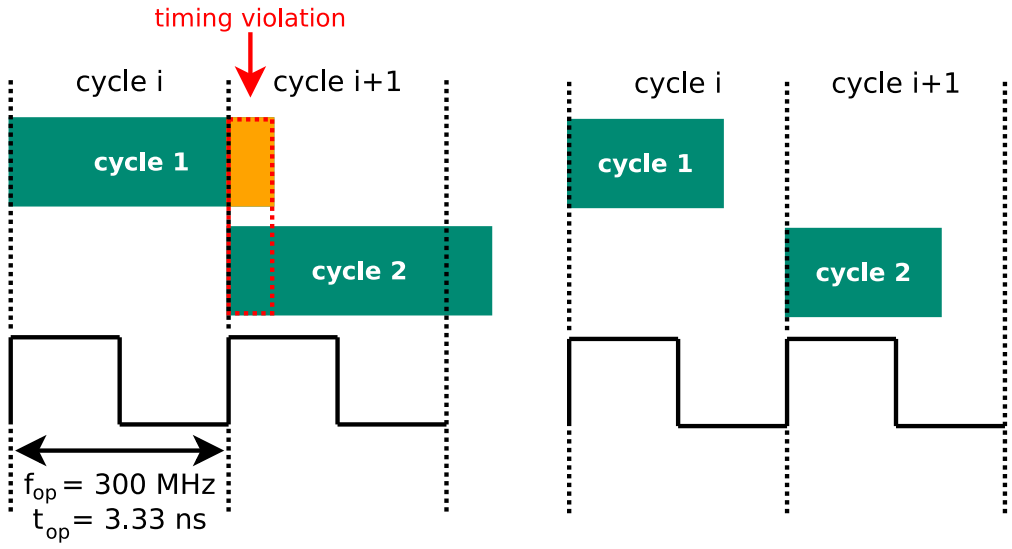
Prior to any DDDG manipulation, the initial timing analysis adjusts each FU on the hardware profile library to have the lowest latency configuration where t_{cp} still fits the timing budget. The DDDG is then annotated with the latencies of the selected configurations, and ASAP/ALAP scheduling is used to define the preferred schedule window of each node. Figure 14 presents an example of FU configuration to avoid timing violation.

[4] l_{min} or l_{max} do not refer to the minimum and maximum latencies, respectively, but to the latencies at the minimum and maximum periods.

Algorithm 2 – The FU characterisation algorithm

```
1:  ε ← 0.001                                          ▷ Value used as recursive break
2:
3:  FU ← fadd                                          ▷ Select the FU to explore
4:  (l_lmin, t_lmin) ← HLSRUN(FU, 2.0)                 ▷ 2ns period, 500MHz
5:  (l_lmax, t_lmax) ← HLSRUN(FU, 60.0)                ▷ 60ns period, 16.66MHz
6:
7:  if l_lmin = l_lmax then
8:      return {(l_lmin, t_lmin)}    ▷ Same latency at bounds, no need to explore in between
9:  else
10:     C ← {(l_lmin, 2.0), (l_lmax, 60.0)}
11:     EXPLORE(FU, (l_lmin, 2.0), (l_lmax, 60.0), C)  ▷ Start recursion
12:     return C
13: end if
```

Figure 14 – Two `fadd` configurations considered at a target clock frequency of 300MHz.



fadd 8 cycles (t_{cp} = 3.45 ns)

(a) This configuration ($l = 8$) causes a timing violation since its critical path delay t_{cp} exceeds t_{op}.

fadd 10 cycles (t_{cp} = 2.46 ns)

(b) Consequently, a configuration with $l = 10$ must be used with a smaller t_{cp} instead, at the cost of taking more clock cycles to complete the operation.

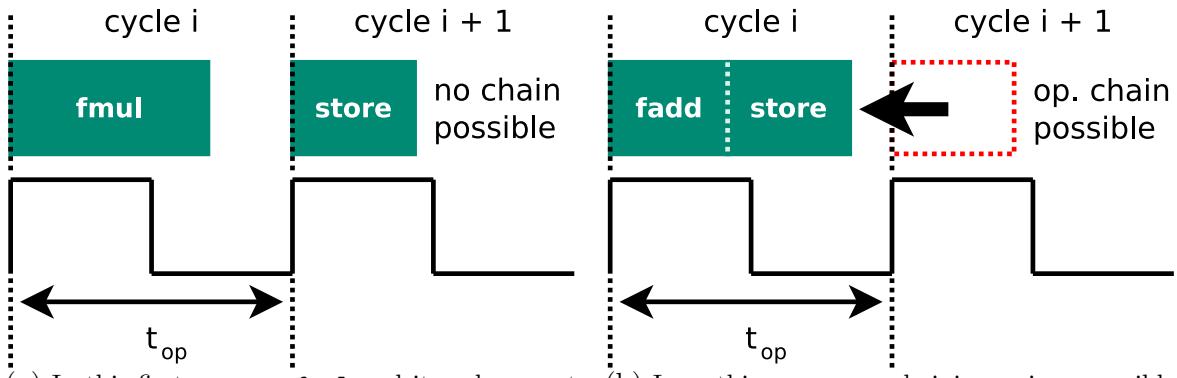
Source: Adapted from Perina *et al.* (2021).

3.2.1.3 *Operation Chaining*

The operation chaining is performed during the resource-constrained scheduling. The scheduler performs three steps to allocate the operations at each clock cycle: it sorts the nodes that are ready for execution using the ASAP/ALAP results as a priority criterion; then it attempts to allocate all ready nodes to the physical resources (e.g. FUs or memory ports); and whenever a node finishes execution, it deallocates the resource that was being used and it marks all the children nodes as ready. With operation chaining, the

steps described above are repeated more than once for every clock cycle scheduling. For example, suppose two operations `a` and `b` are directly dependent (e.g. a multiply followed by the store of its result). In this case, they can be scheduled to the same clock cycle if the accumulated critical path delay of both operations t_{cp}^{a} and t_{cp}^{b} does not exceed the timing budget ($t_{cp}^{a} + t_{cp}^{b} \leq t_{op}$). Figure 15 presents examples of operation chaining attempts. A more detailed flowchart of the resource and timing-constrained scheduling is presented in Appendix D.

Figure 15 – Attempts of operation chaining.



(a) In this first case, an `fmul` and its subsequent store cannot be scheduled to a single clock cycle as they together exceed the timing budget.

(b) In this case, chaining is possible since both `fadd` and its respective store fit in the same clock cycle.

Source: Adapted from Perina *et al.* (2021).

In order to perform the timing budget analysis required for operation chaining, the accumulated critical path delays of all dependent operations scheduled to a single clock cycle must be known. Our first approach to calculate these accumulated delays consisted of maintaining a set of dependency paths P that contained all the scheduled operations within a clock cycle and the dependencies between them. Lina then used P to evaluate whether a candidate instruction could be scheduled in the same clock cycle without violating timing budget.

Algorithm 3 presents this former function, where n is the candidate node for being scheduled, P is the set of all active paths within a clock cycle, and t_{op} is the operating clock period[5]. The `areConnected(p, q)` function returns true if node q depends on p, and `APPENDNEWPATH(P, p)` appends a new path p to the set of paths P.

This first approach, however, does not scale well with kernels that have many independent paths within a clock cycle. For these cases, P might reach very large sizes and the time spent by Lina to perform the timing budget analysis becomes non-negligible. Some of the AES kernels used in this thesis to validate Lina fall on this category. Since

[5] Relating to Figure 58, the node "Does any path violate the timing budget?" would be represented by lines $5-17$ from the algorithm.

Algorithm 3 – Former approach to verify if in-clock cycle scheduling is possible

```
1: procedure validateTimingScheduling(P, t_op, n)
2:     P' ← P
3:     pathFound ← false
4:
5:     for all p ∈ P do                          ▷ For all paths in the path set
6:         for i ← 1, |p| do                     ▷ Iterate through all nodes in the path
7:             if areConnected(p_i, n) then
8:                 pathFound ← true
9:                 p' ← {p_0, p_1, ..., p_i, n}
10:                appendNewPath(P', p')
11:
12:                if (t_p0 + t_p1 + ... + t_n) > t_op then
13:                    return false               ▷ Timing violation
14:                end if
15:            end if
16:        end for
17:    end for
18:
19:    if ¬ pathFound then
20:        appendNewPath(P', {n})   ▷ No deps. found. Create new path with only n
21:    end if
22:
23:    P ← P'                                    ▷ Commit the new paths
24:    return true                               ▷ No violations found
25: end procedure
```

the scheduling model of Lina does not perform any type of backtrack (i.e. operations that are already scheduled cannot be un-scheduled), the critical path delay of scheduled dependency paths becomes invariant, discarding the need to maintain P. Thus, the accumulated critical path delay $d(\cdot)$ up to a candidate node n_i is found simply by adding its delay t_{n_i} to the largest accumulated delay of all its k direct scheduled predecessors n_{i-1}^k:

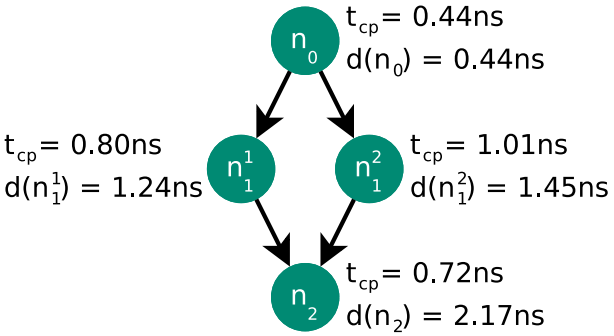
$$d(n_i) = t_{n_i} + \max\{d(n_{i-1}^1), d(n_{i-1}^2), ..., d(n_{i-1}^k)\} \tag{3.3}$$

For nodes n_0 that either have no predecessors or all were scheduled in previous cycles, its largest delay is composed only by its own critical path delay:

$$d(n_0) = t_{n_0} \tag{3.4}$$

Figure 16 exemplifies a set of dependent nodes within a clock cycle along their t_{cp} and $d(\cdot)$ values. Compared to maintaining and calculating the paths in P, this latter approach only requires a `max` operation and one addition per evaluation, which is significantly faster.

Figure 16 – Example of dependent nodes, their critical path delays t_{cp} and largest delays up to each node ($d(\cdot)$).



Source: Elaborated by the author.

3.2.2 Non-Perfect Loop Analyser

The Non-Perfect Loop Analyser implements the DDDG generation and scheduling process as previously explained for multiple code segments inside a loop nest. Such feature allows a broader range of code patterns to be supported by Lina with increased accuracy.

A loop nest is considered non-perfect if there are operations located outside of the innermost loop body, but still within the nest. Figure 17 presents two examples of non-perfect loops. The highlighted regions are composed of statements that are placed before, inside, and after loop levels.

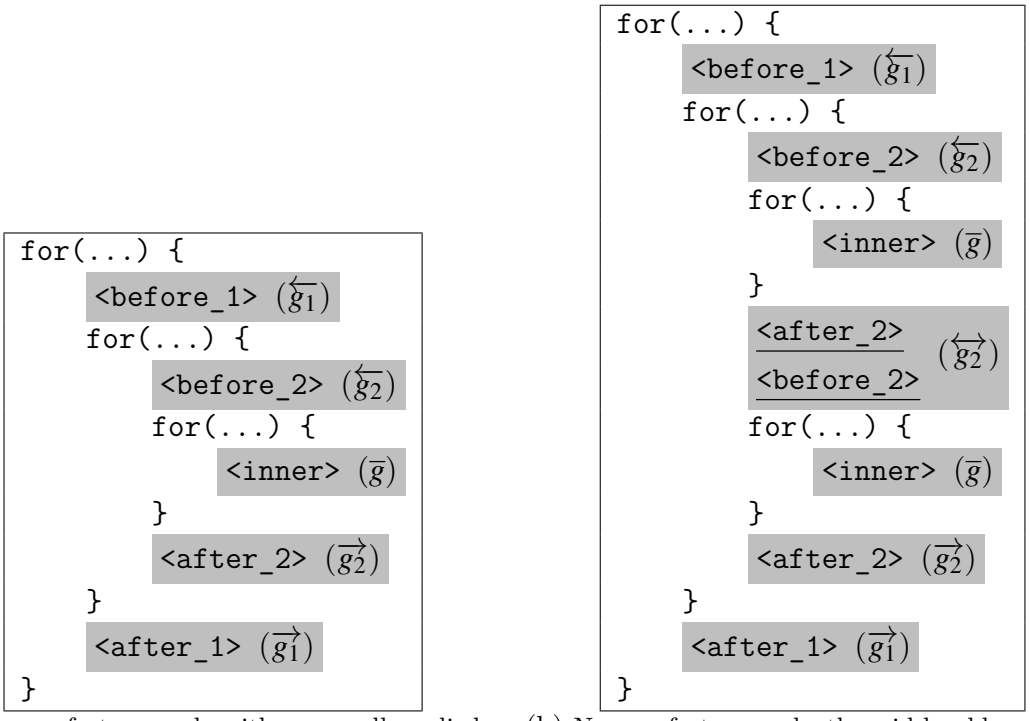
Lina considers four different types of DDDGs for its non-perfect loop model:

- `normal` (or `inner`): innermost loop body (also used for perfect loop nests);
- `before`: region located right before the start of a subloop;
- `after`: region located right after the end of a subloop;
- `between`: region generated by the merge of `before` and `after` DDDGs in unrolled loops.

The `between` type is a particular case present in unrolled loop codes. For example, Figure 17b is the code of Figure 17a with the mid-loop level unrolled by a factor of 2. The `before` DDDG of the replicated body is placed right after the `after` DDDG of the first body (both are underlined on the figure). These two regions can be scheduled together, and thus Lina generates a single `between` DDDG that merges both regions.

For clarity in the equations, we represent each type of DDDG through a notation as indicated in Figure 17. Using an arbitrary metric g as an example, the variables $\overleftarrow{g_i}$, $\overrightarrow{g_i}$ and $\overleftrightarrow{g_i}$ represent g for the `before`, `after` and `between` DDDGs at the i-th loop level,

Figure 17 – Examples of non-perfect loop nests. Shaded regions represent the group of statements that are placed before, inside, and after each loop level. A separate DDDG g is generated for each region. The notation used to identify the DDDGs generated according to each region is shown in parentheses.



(a) Non-perfect example with no unroll applied.

(b) Non-perfect example, the mid-level loop is unrolled with a factor of 2.

Source: Adapted from Perina *et al.* (2021).

respectively. For normal DDDGs, we omit the loop level (i.e. $\overline{g}$) as they always represent the innermost loop body.

Although the operations inside each region are scheduled for parallelism as explained in the previous sections, we assume that the regions themselves are serially executed following the normal software flow, as performed by Vivado HLS. For example in Figure 17b, one iteration of the mid-level loop executes in the following order: $\overleftarrow{g_2}$, $\overline{g}$ (multiplied by the inner loop trip count), $\overleftrightarrow{g_2}$, $\overline{g}$ (multiplied by the inner loop trip count), and $\overrightarrow{g_2}$.

The total cycle count c of a loop nest is calculated recursively. Given a loop nest with K levels, the recursion starts at loop level L and traverses outwards until the outermost loop is reached. The loop level L used as starting point varies according to the presence of loop pipeline, as follows:

- If no pipeline is enabled in any loop level, the starting point is the innermost loop level ($L = K$). The starting cycle count c_L is given by:

$$c_L = c_K = \overline{g} \cdot b_K \cdot u_K^{-1} + \sigma \tag{3.5}$$

where $\overline{g}$ is the scheduled cycle count of the innermost DDDG, b_K and u_K are the bound and unroll factor of the innermost loop, respectively, and σ is a constant that represents the cycles spent on testing the enter and exit conditions of a loop;

- If pipeline is enabled for loop level P ($P \in \{1,\ldots,K\}$), the recursion starts from level P instead ($L = P$). In this case, the starting cycle count c_L is given by an equation derived from Equation 3.1:

$$c_L = c_P = MII \cdot (b_P \cdot u_P^{-1} - 1) + \overline{g} + \sigma \tag{3.6}$$

where MII is the value given by Equation 3.2. When pipeline is enabled for a certain loop level, all its subloops are automatically fully unrolled by Vivado HLS. As a consequence, the loop nest is virtually pruned to P levels due to flattening caused by the unrolls. Lina replicates this behaviour by including all operations from loop levels between P and K in a single DDDG. In this case, $\overline{g}$ includes the scheduling of all automatically unrolled subloops.

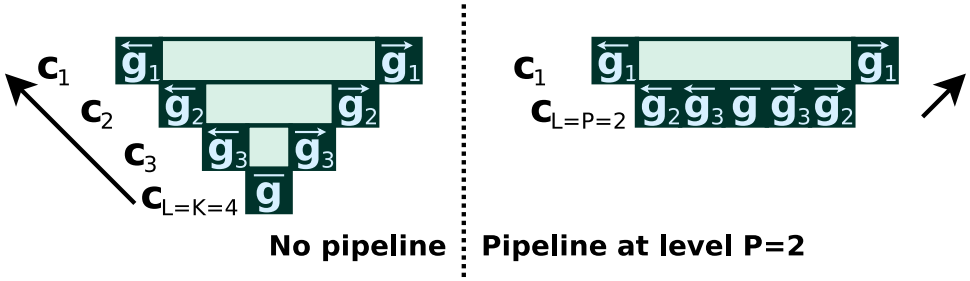
With c_L calculated, the cycle count c_l of the remaining loop levels l ($l < L$) considers the accumulated cycle count of its subloop c_{l+1} along with the cycle count of non-perfect segments for the current loop level ($\overleftarrow{g_l}$, $\overrightarrow{g_l}$ and $\overleftrightarrow{g_l}$):

$$c_l = (\overleftarrow{g_l} + \overrightarrow{g_l} + (\overleftrightarrow{g_l} \cdot (u_l - 1)) + (c_{l+1} \cdot u_l)) \cdot b_l \cdot u_l^{-1} + \sigma \tag{3.7}$$

where b_l and u_l are the bound and unroll factor of loop level l, respectively. As exemplified in Figure 17, `between` DDDGs are only calculated when unroll is enabled.

Finally, the total cycle count c is obtained when the topmost level is reached ($l = 1$). Figure 18 presents a graphical example of this calculation.

Figure 18 – Abstract example of a non-perfect loop nest ($K = 4$ loop levels) and how the c_l values are calculated. At left, there is no pipeline enabled and thus the calculation starts from the innermost level ($L = K = 4$). At right, pipeline is enabled for the loop level $P = 2$ and thus the calculation starts from $L = P = 2$.



Source: Elaborated by the author.

3.2.3 Resource Awareness

The following subsections present the resource model implemented by Lina, which considers floating-point and integer datapaths, memory-related resources (e.g. distributed RAM arrays, completely partitioned arrays, load/store registers), intermediate registers (e.g. buffers, pipeline registers) and extra resources used by auxiliary computations (e.g. loop/array indexing).

3.2.3.1 Functional Unit Resource Estimation

Modern HLS compilers often balance the amount of FUs to be synthesised based not only on the scheduling, but also on the routing logic generated to supply and retrieve the different inputs and outputs. An extensively reused FU might require an overly complex multiplexing logic that can use more resources than the FU itself.

During the RCLS step, Lina calculates the amount of FUs required to execute each DDDG separately. Two different approaches based on FU's complexity are used to calculate the total amount considering all DDDGs.

The first approach considers that simple FUs (e.g. with a small resource footprint) are less likely to be reused. In this case, we accumulate the resources of the DDDGs in the loop nest. We use this approach for integer and logic operations:

$$FU_q = \overline{FUc}_q + \sum_l \max\{\overleftarrow{FUc}^l_q + \overrightarrow{FUc}^l_q, \overleftrightarrow{FUc}^l_q\} \tag{3.8}$$

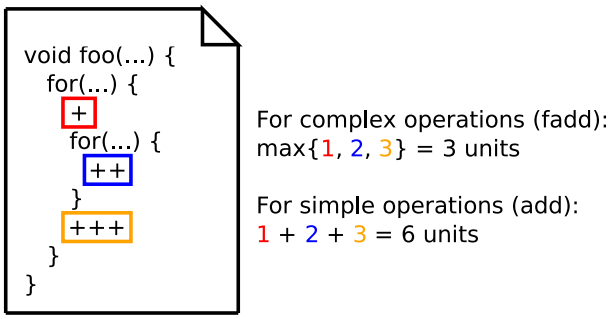
where q is the type of FU (e.g. `add`, `mul`), l is the loop level, and FUc_q is the constrained amount of FUs of type q estimated by RCLS. The `between` DDDG is equivalent to the logic of `before` and `after` together. Therefore, we consider the largest number of FUs from the options {`before` + `after`, `between`} for every loop level (hence the max operation).

The second approach considers that resource-hungry FUs are more likely to be reused. In this case, the HLS compiler prefers to extensively reuse them before replicating. We assume that these FUs are always shared, and the total amount is given by the DDDG with the largest usage. This approach is used for floating-point operations, as follows:

$$FU_q = \max_l\{\overline{FUc}_q, \overleftarrow{FUc}^l_q, \overrightarrow{FUc}^l_q, \overleftrightarrow{FUc}^l_q\} \tag{3.9}$$

An example for both calculations is presented on Figure 19.

Figure 19 – Two examples of FU calculation based on the RCLS allocation results FUc_q of each DDDG (each + denotes one FU allocated). The first calculation considers these adders as floating-point (complex) and the other considers as integer (simple).



Source: Elaborated by the author.

We consider the execution of a single iteration when calculating $\overline{FUc_q}$. Since multiple iterations are allowed to overlap when pipeline is enabled, $\overline{FUc_q}$ might underestimate the amount of resources required. In this case, we use a different approach based on Zhao *et al.* (2019), Gao, Wickerson and Constantinides (2016), where the the $\overline{FUc_q}$ used in Equation 3.8 and Equation 3.9 is calculated as follows:

$$\overline{FUc_q} = \left\lceil \frac{\overline{N}_q}{MII} \right\rceil \tag{3.10}$$

where $\overline{N}_q$ is the number of nodes of type q in the `normal` DDDG and MII is the value given by Equation 3.2. The rationale behind this approach is that the number of overlapped iterations increases with the decrease of the initiation interval, which in turn increases the demand for resources required to execute multiple iterations at once. For example, if the initiation interval is 1, all nodes of the DDDG will be active for different iterations at every clock cycle when the pipeline is filled. Therefore $\overline{N}_q$ FUs are required so that each node of type q can be simultaneously active.

The total number of resources is calculated by accumulating the products between the resources required to instantiate a single FU of type q (i.e. $LUT_q(f_{op})$, $FF_q(f_{op})$, and $DSP_q(f_{op})$) and the number of instantiated FUs from that type (FU_q), as presented in Equation 3.11:

$$LUT_{fu} = \sum_q \{FU_q \cdot LUT_q(f_{op})\} \tag{3.11a}$$

$$FF_{fu} = \sum_q \{FU_q \cdot FF_q(f_{op})\} \tag{3.11b}$$

$$DSP_{fu} = \sum_q \{FU_q \cdot DSP_q(f_{op})\} \tag{3.11c}$$

The amount used by a single FU (e.g. $LUT_q(f_{op})$) is dependent on the target frequency f_{op}, as explained in subsection 3.2.1. No BRAM is used by any type of FU that we consider.

3.2.3.2 Array-related Resource Estimation

Arrays can be either located in on-chip or off-chip memory. Lina estimates the resource only for arrays located in on-chip memory, as presented in this subsection. This does not affect estimation, however, since our resource-aware objective is contained to on-chip FPGA resources.

For each array m in the code, we calculate the LUTs, FFs and BRAMs required (LUT_m, FF_m and $BRAM_m$, respectively) according to three different scenarios as explained below. No DSPs are needed by array-related modules.

3.2.3.2.1 Scenario I

An array is stored using LUTs as SRAM memories when its size is smaller than a specific threshold and no complete partitioning is performed. Thus LUTs are used for storage, FFs for input/output buffers, and no BRAM modules are used, as follows:

$$LUT_m = p_m \cdot \left\lceil \frac{ps_m}{64} \right\rceil \tag{3.12a}$$

$$FF_m = p_m \cdot (\lceil \log_2\{pn_m\} \rceil + ws_m \cdot \lambda_m) \tag{3.12b}$$

$$BRAM_m = 0 \tag{3.12c}$$

$$ps_m = pn_m \cdot ws_m \tag{3.12d}$$

where p_m is the partition factor, ps_m is the total size of a partition in bits, pn_m is the number of words within a partition, ws_m is the word size in bits and λ_m is 1 when m is a read-only array, 2 otherwise.

The constant values were inferred from the analysis of several Vivado HLS early resource reports.

3.2.3.2.2 Scenario II

An array is stored using BRAM modules if it exceeds the threshold used in scenario I and if no complete partitioning is performed. No LUTs are used, and FFs are used for input/output buffers. We calculate the number of BRAMs required for each array partition as being the smallest power of 2 that can still fit the partition size, similar to Zhong *et al.* (2016) and Makni *et al.* (2018):

$$LUT_m = 0 \tag{3.13a}$$

$$FF_m = p_m \cdot \lceil \log_2\{pn_m\} \rceil \tag{3.13b}$$

$$BRAM_m = p_m \cdot \min_{i \in \mathbb{N}_0}\{2^i : 2^i \geq \frac{ps_m}{\beta}\} \tag{3.13c}$$

where β is the total size of a BRAM module in bits, which depends on the target FPGA. Lina considers a size of 18 kbits for the platforms supported.

3.2.3.2.3 Scenario III

In the last scenario, a fully-partitioned array stores all its elements in FFs instead of BRAM modules for full parallel access. In this case, we use the following equations:

$$LUT_m = 0 \tag{3.14a}$$

$$FF_m = ts_m \tag{3.14b}$$

$$BRAM_m = 0 \tag{3.14c}$$

where $ts_m = p_m \cdot ps_m$ is the total size of array m in bits.

No LUTs and BRAM modules are used in this case.

3.2.3.2.4 Total Array Resource Usage

With LUT_m, FF_m and $BRAM_m$ calculated for each array m according to the scenarios above, we can calculate the total LUT, FF and BRAM usage of array-related resources as follows:

$$LUT_{mem} = \sum_m \{LUT_m\} \tag{3.15a}$$

$$FF_{mem} = \sum_m \{FF_m\} \tag{3.15b}$$

$$BRAM_{mem} = \sum_m \{BRAM_m\} \tag{3.15c}$$

3.2.3.3 Complete Resource Estimation

Finally, we integrate the LUT and FF calculations from Makni *et al.* (2018) in order to provide a more comprehensive estimation. The total LUT, FF, DSP and BRAM usage is given by the equations below:

$$LUT_{total} = LUT_{fu} + LUT_{mem} + LUT_{mux} + LUT_{ex} \tag{3.16a}$$
$$FF_{total} = FF_{fu} + FF_{mem} + FF_{reg} \tag{3.16b}$$
$$DSP_{total} = DSP_{fu} \tag{3.16c}$$
$$BRAM_{total} = BRAM_{mem} \tag{3.16d}$$

where LUT_{fu}, FF_{fu}, DSP_{fu}, LUT_{mem}, FF_{mem} and $BRAM_{mem}$ are the FU and array related resources as previously explained; LUT_{mux} and LUT_{ex} are the LUT usage of multiplexers and auxiliary computations (e.g. loop/array indexing), respectively; and FF_{reg} is the number of FFs allocated to the intermediate registers.

The following items describe the calculations from Makni *et al.* (2018) that we use to calculate LUT_{mux}, LUT_{ex}, and FF_{reg}, and the rationales behind each of them. For a single loop nest with K levels, consider the following:

- B is the product of all loop level bounds:

$$B = \prod_{l} \{b_l\}, \quad l \in \{1, 2, \ldots, K\} \tag{3.17}$$

- U is the product between the unroll factors of all loop levels in the nest, and U_i is the product of unroll factors between loop level i and top-level (i.e. $U = U_K$):

$$U_i = \prod_{l} \{u_l\}, \quad l \in \{1, \ldots, i\}, i \leq K \tag{3.18}$$

- e is an exponential constant derived from B:

$$e = \min_{i \in \mathbb{N}_0} \{2^i : 2^i \geq B\} \tag{3.19}$$

  Rationale: Makni *et al.* (2018) assume that there is a relation between the loop bound B and a simple exponential parameter e in the form of $B \in O(2^e)$. Thus, we select the smallest integer value of e so that $2^e \geq B$;

- $V_1 = e + 1$, $V_2 = 2 \cdot e$ and $V_3 = e + 2$ are three exponential parameters derived from e;

- N_{load} and N_{store} are the total number of load/store operations in a loop nest compensated by unroll:

$$N_j = \overline{n}_j \cdot U_{K-1} + \sum_{l} \{(\overleftarrow{n}^l_j + \overrightarrow{n}^l_j + (\overleftrightarrow{n}^l_j \cdot (u_l - 1))) \cdot U_{l-1}\} \tag{3.20}$$

  where $j \in \{load, store\}$ and the n_j values represent the number of loads/stores present in the DDDGs ($l \in \{1, \ldots, K-1\}$).

Rationale: Makni *et al.* (2018) consider that N_{load} and N_{store} represent the number of loads and stores identified in the single DDDG graph used in their model. We consider that unroll replicates the number of loads and stores. Therefore we multiply n_j by the accumulated unroll factors down to the loop level where each DDDG is located. For the `normal` DDDG, U_{K-1} instead of U_K is used since $\overline{n}_j$ already considers the innermost loop unroll (see Zhong *et al.* (2016)). For the remaining DDDGs, U_{l-1} is used as a multiplying factor to consider the number of times that they are repeated due to nested unroll. Additionally, the `between` DDDG is multiplied by $(u_l - 1)$ because each time its loop level is unrolled, there are $u_l - 1$ regions in the replicated code represented by this DDDG.

- C is the accumulation of the iteration latencies of each DDDG:

$$C = \overline{c}_K \cdot U_{K-1} + \sum_l \{(\overleftarrow{c}_l + \overrightarrow{c}_l + (\overleftrightarrow{c}_l \cdot (u_l - 1))) \cdot U_{l-1}\} \tag{3.21}$$

Rationale: Makni *et al.* (2018) consider that IL represents the number of cycles needed to perform a single iteration of the loop. Using a similar rationale as N_{load} or N_{store}, the IL (here named C) is adapted to consider non-perfect loop nests.

- N_{op} is the total number of FUs allocated of all types:

$$N_{op} = \sum_q \{FU_q\} \tag{3.22}$$

Using the parameters above, the estimated LUT amounts for multiplexers LUT_{mux} and auxiliary computations LUT_{ex} are given by:

$$LUT_{mux} = 32 \cdot (N_{store} + N_{op}) + K \cdot V_1 + 14 \cdot N_{load} \tag{3.23a}$$

$$LUT_{ex} = K \cdot (V_1 + V_2 + V_3) + (U - 1) \cdot V_1 \tag{3.23b}$$

The estimated amount of FFs related to intermediate registers FF_{reg} is given by:

$$FF_{reg} = 32 \cdot (N_{load} + N_{store} + N_{op}) + C + K \cdot V_1 \cdot \gamma \tag{3.24a}$$

$$\gamma = \begin{cases} 1 & \text{if } K = 1 \\ 2 & \text{otherwise} \end{cases} \tag{3.24b}$$

Parts of the equations from Makni *et al.* (2018) that consider more than one loop were simplified since only a single loop nest is currently considered here. The γ value, for example, is a simplification of the S_l and L_K values from Makni *et al.* (2018, eq. (8)).

3.2.4 Off-chip Memory Model

The FPGA accesses the input/output data from two memory spaces: on-chip and off-chip. On-chip memory spaces are constructed using internal FPGA resources as shown in the previous section. Off-chip memory spaces are usually composed of SDRAM/SRAM memory modules located outside of the FPGA die. By default, Lina considers that all arrays are stored in on-chip memory components.

Although on-chip memory modules provide low-latency access (1 to 2 cycles), they are usually limited in size, as compared to large arrays that often compose large-scale applications. In this case, the on-chip resources should either be used as small buffers, or the data should be directly accessed from the off-chip space.

The off-chip memory modules have larger read/write latencies than compared to BRAM modules and their performance is intimately linked to the data access pattern. To this end, Lina leverages the generated dynamic trace to infer the memory accesses. Then, it uses an off-chip memory model that analyses the memory patterns, attempting to identify memory optimisation opportunities and reports potential bottlenecks.

To activate the off-chip memory model, the user must indicate to Lina which arrays should be considered off-chip. Then, for each of those arrays, Lina transforms the original DDDGs and swaps the load/store nodes from the DDDG with the respective off-chip counterparts. Several analyses are then performed in search for potential off-chip optimisations, such as coalescing. Finally, the DDDGs are scheduled and constrained as usual, however the memory model is consulted before any off-chip transaction scheduling (for example to delay overlapping memory instructions).

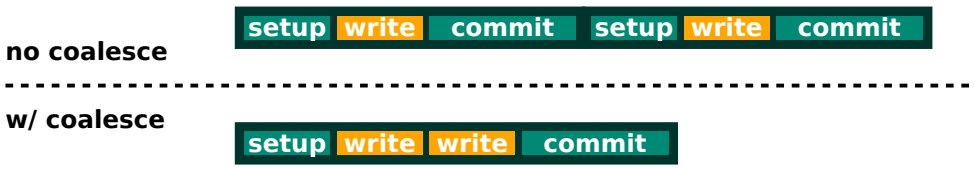
For the memory model here presented, we consider that off-chip memory transactions are shaped around three steps: `setup`, `action` and `commit`. The `setup` step includes operations that are required to prepare the off-chip memory system for read/write. Then, the `action` step is where an actual read or write is performed. This step can be repeated multiple times depending on the memory pattern, like coalescing. Then, some additional final time might be required to finish the memory transaction, which is the `commit` step. Not every transaction requires `setup` and `commit` simultaneously, some interfaces might require only one of these steps. The number of clock cycles c^{mem}_{total} required to perform an off-chip memory transaction can be modelled as:

$$c^{mem}_{total} = c^{mem}_{setup} + (n \times c^{mem}_{action}) + c^{mem}_{commit} \tag{3.25}$$

where c^{mem}_{setup}, c^{mem}_{action} and c^{mem}_{commit} are the number of clock cycles required to perform the `setup`, `action` and `commit` stages, respectively, and n is the number of coalesced read-/writes included in this single transaction. In general, the `setup` and/or `commit` steps are time consuming as compared to the `action` steps. Therefore increasing n — and thus

performing coalesced transactions — is crucial for efficient off-chip memory use. As an example, Figure 20 presents two cases of off-chip read/write transactions, without and with coalescing, respectively.

Figure 20 – Two cases of off-chip transactions, one without coalescing (top) and with coalescing (bottom).



Source: Elaborated by the author.

The FPGA platform and HLS compiler toolchain that we use in this chapter — Xilinx Zynq UltraScale+ and Xilinx SDSoC toolchain, respectively — maps off-chip memory accesses in a similar fashion as presented above. Chart 2 presents the instructions that compose read and write transactions from C/C++/OpenCL kernels when using this toolchain and how they relate to the transaction steps previously described. We also present the number of clock cycles required to execute each instruction considering the platform that we use (ZCU104) as an example. Note that there is no `commit` phase for read transactions.

Chart 2 – Instructions executed by SDSoC/Vivado-generated designs for off-chip access, their relation to the abstract transaction steps, and the cycle count of each instruction in the ZCU104 platform.

Instruction name	Transaction step	# cycles ZCU104
Read transaction		
Read request (`ReadReq`)	`setup`	134
Read (`Read`)	`action`	1
Write transaction		
Write request (`WriteReq`)	`setup`	1
Write (`Write`)	`action`	1
Write response (`WriteResp`)	`commit`	132

Source: Elaborated by the author.

In the next sections, we explain the features supported by our memory model, including HLS limitations and how they affect performance. We then present how unroll and pipeline affects the memory scheduling. Finally, we present the memory report generated by Lina that includes optimisation suggestions and warnings about potential bottlenecks.

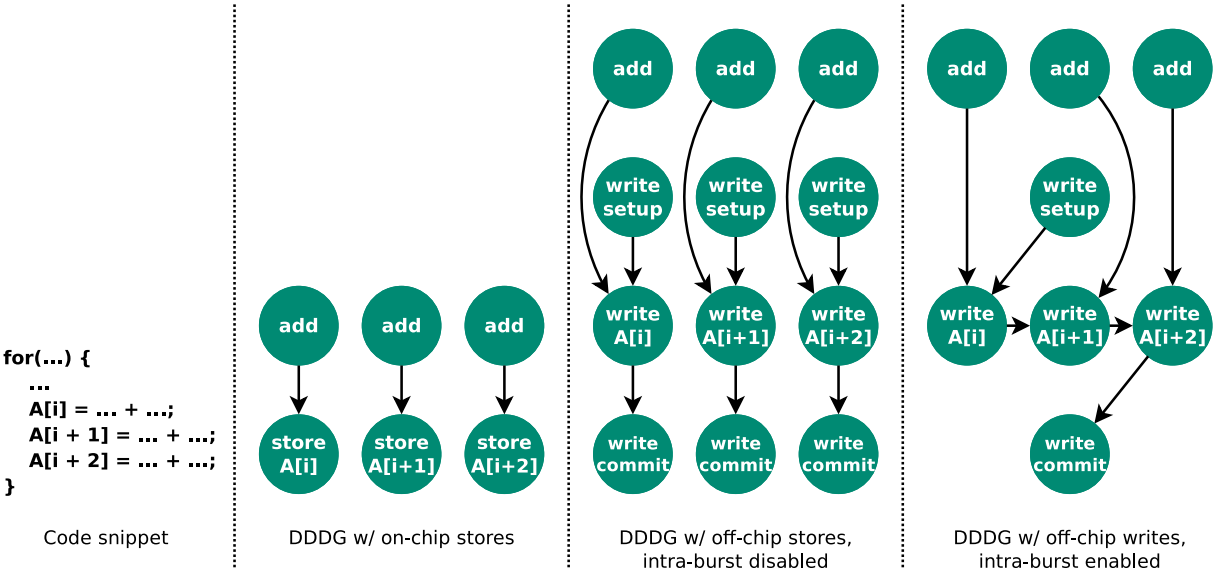
3.2.4.1 Memory Model Features and Behaviour

The model implements four optimisation features, presented below:

3.2.4.1.1 Intra-iteration bursts

If there are multiple reads or writes within a DDDG and they are all sequentially addressed, Lina allows them to be merged into a single memory transaction. This means that the `setup` and `commit` steps are shared among all of these merged read/write accesses. Figure 21 presents an example code and how enabling or disabling intra-iteration burst affects the DDDG transformation.

Figure 21 – Examples of DDDGs for a simple loop. Please note that although there is no DDDG dependency between the three transactions when intra-burst is disabled, they might be constrained during the RCLS phase and overlapping is dependent on certain conditions (please see subsubsection 3.2.4.2 for more details).



Source: Elaborated by the author.

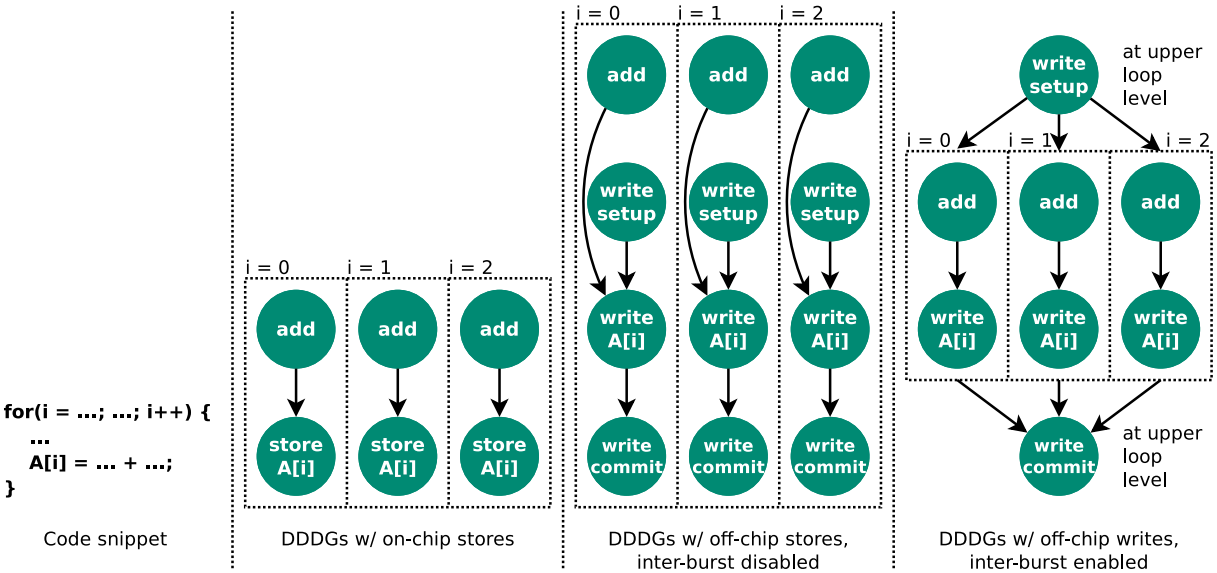
3.2.4.1.2 Inter-iteration bursts

The intra-iteration burst feature as just explained may reduce the overall latency of the design, since less `setup` and `commit` steps are performed, and existing ones are reused by multiple read/writes. However, these steps are still contained within a loop iteration, meaning that they execute at every iteration.

The inter-iteration burst feature performs the coalescing analysis as previously explained, but across multiple sequential iterations of a loop. If such pattern is found, a single `setup` and a single `commit` steps are performed before and after the loop where the read/writes are located, respectively. This effectively removes these time-consuming

steps from inner loops, drastically reducing the overall design latency in some cases. Figure 22 presents an example code and how toggling inter-iteration burst affects the DDDG transformation.

Figure 22 – Examples of DDDGs for a simple loop. DDDGs representing multiple iterations are presented to exemplify the relation to the `setup` and `commit` steps, though only one iteration is scheduled. Similar to last figure, the RCLS phase might constrain multiple transactions to not overlap even if their DDDGs are independent.



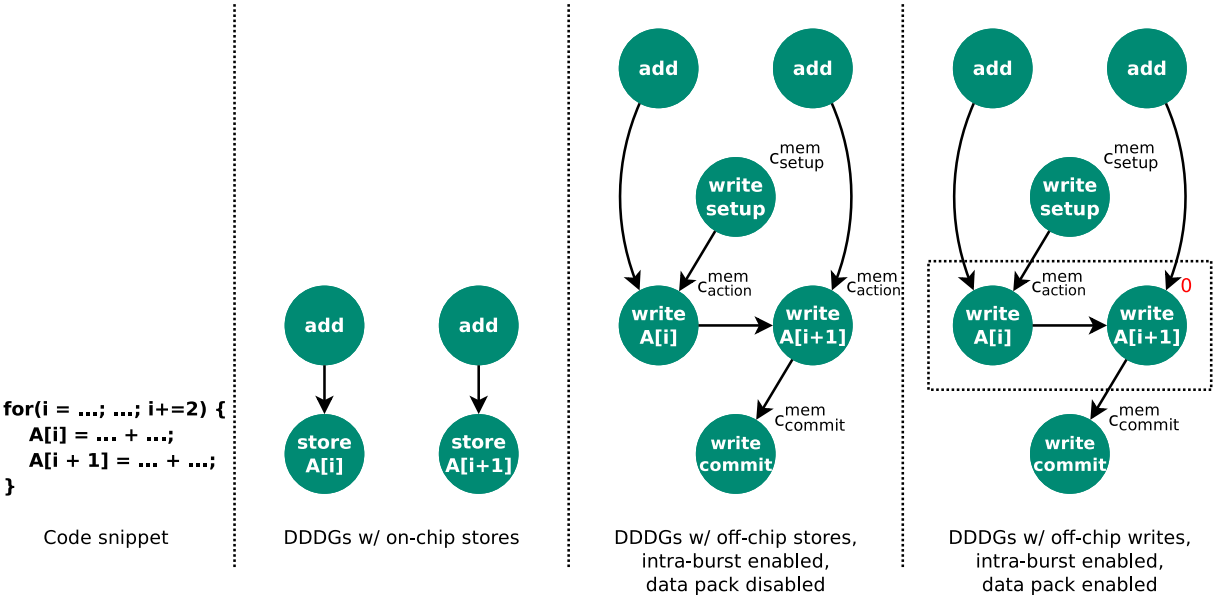
Source: Elaborated by the author.

3.2.4.1.3 Data packing

Some off-chip technologies are coupled with wide data buses that allow more than one array element to be accessed in a single transaction. GDDR5 devices, for example, have a 256-bit wide data bus (Micron Technology, Inc., 2014). Vector data types can be used to utilise these wide buses by accessing multiple elements at once. In OpenCL, for example, a `float4` is a data type containing 4 floating point variables in a single word (128-bit wide).

Lina performs data packing by detecting if the read/write transactions within a DDDG can be modelled as groups of contiguous and equally-sized transactions. For example, if a DDDG contains four contiguous float reads, they can be packed in two `float2` or a single `float4`. However, these data types must be defined at compile-time and are not interchangeable during execution. This means that if Lina decides that `float4` is a possible data pack size for one DDDG, this size must also be valid for all other DDDGs that also use this array. Lina performs the data pack analysis for all DDDGs and then reports which is the largest size compatible with all DDDGs. Figure 23 presents an example code and how data packing affects a DDDG.

Figure 23 – Examples of DDDGs for a simple loop. The off-chip memory nodes are annotated with the number of cycles required to solve each one. In the last (right) case, the data packing analysis of Lina identifies that both writes can be packed together as a single vectorised value. Lina allows both nodes to be scheduled in the same clock cycle by assigning 0 to the latency of the second write (highlighted in red).



Source: Elaborated by the author.

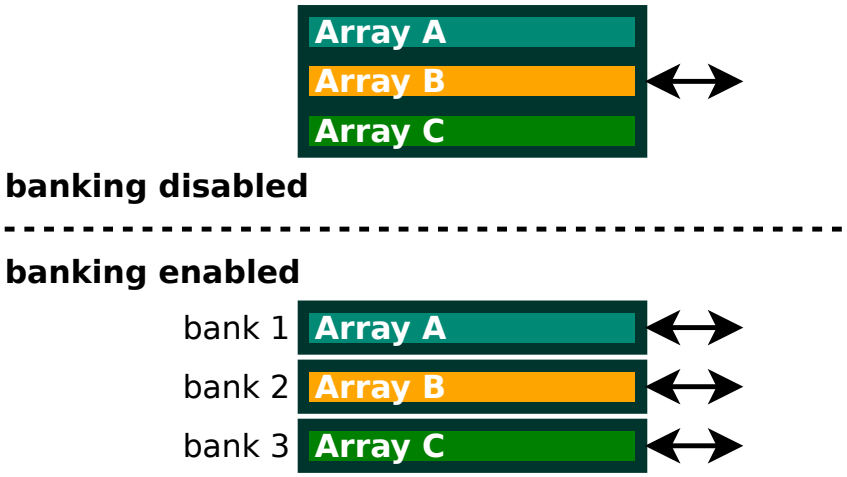
3.2.4.1.4 Memory banking

When assigned to on-chip memory, each array is mapped to a separate BRAM module. Each array has its own ports and are treated separately during resource constraining. On the contrary, off-chip arrays usually share a same bus that communicates to the external modules. In this case, all arrays are treated to be in the same memory space, and constraints like recurrence or port limitations affect the whole memory space.

Some memory controllers provide multiple communication channels that can communicate to separate banks of the off-chip memory module. These can be used in parallel for better memory performance. Lina implements the memory banking feature, on which when enabled, all off-chip arrays are mapped to separate memory banks. In this case, all off-chip arrays are treated to have separate memory spaces, and constraints analyses are restricted to each array similar to the on-chip constraining logic. Figure 24 presents an example of memory banking.

For the following sections, consider that all off-chip arrays share the same memory space if banking is disabled, and that each off-chip array has their own memory space if banking is enabled. Other combinations (e.g. shared memory space only by some arrays) are not considered in this thesis.

Figure 24 – Example of memory space division between three arrays when banking is disabled (top) and enabled (bottom), respectively. The arrows indicate available read/write interfaces.



Source: Elaborated by the author.

3.2.4.1.5 Port management

Initially, the memory model transforms the DDDGs as explained above and prepares auxiliary data structures. Then, the memory model is consulted before any off-chip DDDG node scheduling during RCLS. In this part, the memory model uses the generated auxiliary information to decide whether the scheduling of the candidate off-chip node violates any other ongoing memory transaction or not (e.g. two concurrent writes for same memory space). Nodes that cannot be scheduled in the current clock cycle are delayed until the constraints are lifted.

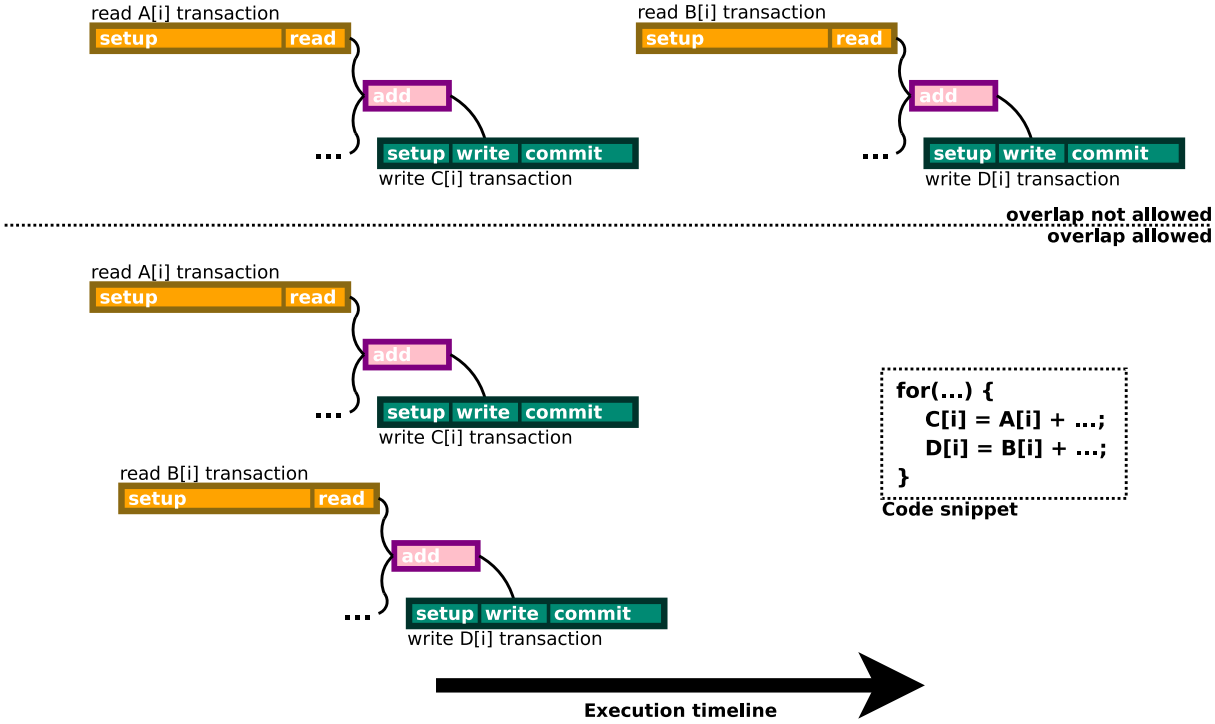
3.2.4.2 Interaction Between Multiple Transactions and Memory Model Policies

When multiple off-chip transactions are to be scheduled, the memory model must evaluate whether they can be overlapped or not. This depends on several aspects, such as if their memory spaces overlap. Figure 25 presents an example where two transactions may or may not overlap, and how this affects the overall schedule.

The decision making related to multiple transactions is also dependent on the analyses performed by each HLS compiler. In our case, we define these decision making policies based in our target HLS tool, i.e. the Xilinx SDSoC toolchain.

We noticed that certain code patterns affect how Vivado schedules off-chip transactions in a global manner. By testing several different code patterns, we detected two different scheduling policies: conservative and permissive. The first one is more conservative in regard to overlapping multiple transactions, whereas the latter has more relaxed constraints for overlapping. Chart 3 presents both policies and how they affect several aspects of the off-chip scheduler.

Figure 25 – Example of a code snippet with two independent read-add-write sequence of instructions. In the first case (above) the transactions are not allowed to overlap and more clock cycles are required, whereas in the second case (below), the transactions are allowed to overlap.



Source: Elaborated by the author.

Chart 3 – The effect of scheduling policies on the memory model.

	Policy	
Aspect	Permissive	Conservative
Inter-iteration bursts (read)	Allowed if there are no other reads and writes for the same array in the same or a deeper loop level	Allowed if there are no other reads for the same memory space and no writes for the same array on the same or a deeper loop level
Inter-iteration bursts (write)	Allowed if there are no other reads and writes for the same array in the same or a deeper loop level	Allowed if there are no other writes for the same memory space and no reads for the same array on the same or a deeper loop level
Setup of a read transaction (`ReadReq`)	Allowed when active reads and writes in the same memory space are for non-overlapping regions	Allowed when active reads are not burst and writes in the same memory space are for different regions
Setup of a write transaction (`WriteReq`)	Allowed when active reads and writes in the same memory space are for non-overlapping regions	Allowed when there is no other active write in the same memory space
Commit of a write transaction (`WriteResp`)	Always allowed.	Allowed when active reads in the same memory space are for different regions
Scheduling of promoted nodes[†]	All can execute simultaneously if all active transactions in the same memory space are non-overlapping	**For read:** all can execute simultaneously if all active transactions in the same memory space are non-overlapping; **For write:** cannot execute simultaneously with any other transaction in the same memory space

[†] Promoted nodes are the `setup` and `commit` nodes that were promoted from inner loop levels due to inter-iteration burst optimisation.

Source: Elaborated by the author.

Although we could not draw a full conclusion on what triggers one policy or another during the Vivado HLS compilation, we detected at least one code pattern that always switches Vivado HLS to the conservative policy. Considering a loop level, if there is an off-chip read transaction located after an off-chip write transaction to the same memory space, Vivado automatically uses the conservative policy, severely degrading performance. Lina detects if such code pattern happens and alerts the user for the potential performance

degradation. We henceforth call this code pattern "read-after-write". Figure 26 presents an example of such pattern.

Figure 26 – Example of a "read-after-write" code pattern: The `B` array is accessed for read right after a write transaction to the same array (both transactions indicated in red).

```
...
B[i]= A[i] * 5;
C[i] = B[i]+ 2;
...
```

Source: Elaborated by the author.

3.2.4.3 Interaction Between Off-chip Transactions and Pragmas

The presence of off-chip transactions affects certain directives, such as loop unroll and pipeline. This section presents how Lina handles these effects.

3.2.4.3.1 Loop unroll

In general, loop unroll increases performance by allowing more loop iterations to be scheduled together, at the cost of increased resource usage. We noticed that when using SDSoC, unrolling loops with off-chip transactions often lead to severe performance degradation. As an example, Source code 2 presents a simple loop nest without and with unroll (arrays A and B are off-chip). In the version without unroll, SDSoC/Vivado successfully detects inter-iteration bursts for both read and write transactions and the total cycle count for $N = 256$ is 1802. When an unroll of 2 is enabled, the cycle count increases to 69761 (38.7× slowdown) due to the inter-iteration bursts not being detected anymore. Although there is still the possibility for such burst optimisations since all reads and writes are still coalesced through the whole loop, we believe that the code is being unrolled prior to the off-chip memory scheduling within Vivado/SDSoC. The read from `A[i + 1]` is located after the write to `B[i]` in the unrolled code, which triggers a "read-after-write" code pattern that blocks the bursts.

Source code 2 – Rolled and unrolled (factor of 2) examples of a simple loop nest, both arrays are off-chip

```
1: // Before unroll
2: for(int i = 0; i < N; i++) {
3:   B[i] = A[i] * 2;
4: }
5:
6: // After unroll of factor 2
7: for(int i = 0; i < N; i += 2) {
8:   B[i] = A[i] * 2;
9:   B[i + 1] = A[i + 1] * 2;
10: }
```

Source code 3 presents a manually-unrolled version of the previous code, on which all off-chip reads were placed before all off-chip writes. In this case, the HLS scheduling was more permissive and a cycle count of 35073 was reached, which is still slower than the baseline version with no unroll. Although explicitly removing the "read-after-write" pattern brought improvements, Vivado detects that there are two reads for the same array `A` and two writes for the same array `B` in the unrolled code, which disables inter-iteration burst as described in Chart 3. Although all transactions are coalesced, we believe that this information is not being properly handled by Vivado's internal analyses.

Source code 3 – Manually unrolled (factor of 2) variant of the simple loop nest

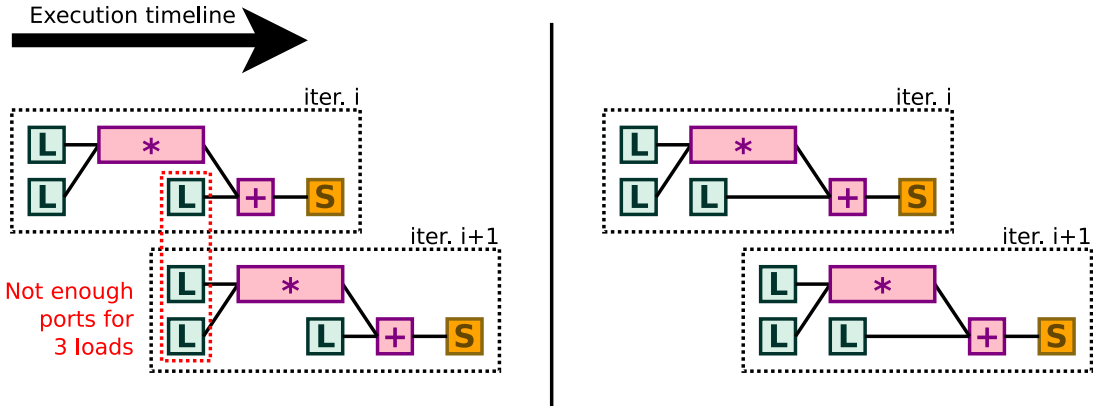
```
1: for(int i = 0; i < N; i += 2) {
2:   float lA = A[i], lAp = A[i + 1];
3:
4:   B[i] = lA * 2;
5:   B[i + 1] = lAp * 2;
6: }
```

3.2.4.3.2 Loop pipeline

The calculations performed in Equation 3.2 are optimistic, as they consider that instructions within a scheduled loop iteration can be moved to earlier or later stages in order to relax pipelining constraints. Memory constraints are not easily solvable without moving instructions or increasing the pipeline's II, since there is no possibility on generating more memory ports for increased concurrency than the existing ones in the BRAM modules. For example, Figure 27 presents two schedules for the same computation. Considering that the arrays have 2 concurrent read ports per cycle, a pipelining with $II = 2$

is not possible due to requiring 3 reads in a single cycle. However, by simply moving one of the load instructions one schedule slot back, it is possible to schedule using $II = 2$.

Figure 27 – Two schedule attempts with $II = 2$. Instructions are represented by small boxes (L and S stand for load and store, respectively). At left, the pipeline is not possible at this II due to requiring more read ports than available. At right, one of the load instructions is moved within the schedule, which in turn lifts the port restriction.



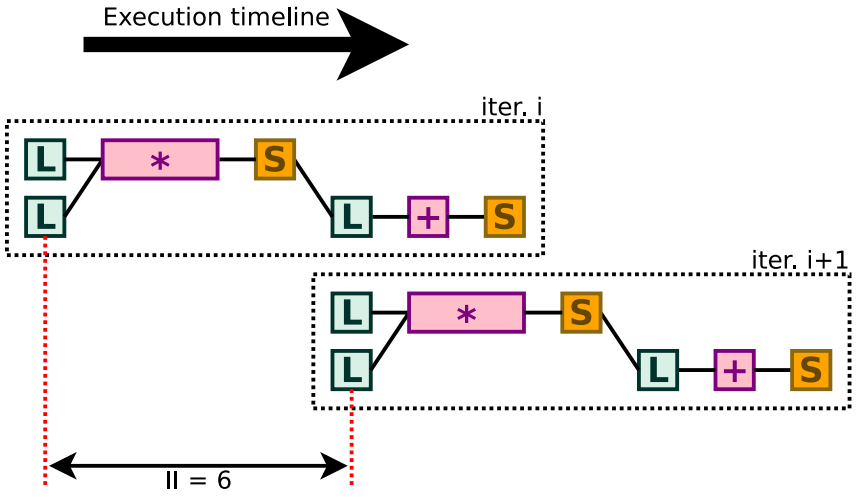
Source: Elaborated by the author.

Load and store nodes are often starting/terminal nodes of a schedule. In this case, they can be easily moved to the beginning or the end of a schedule window. This reduces the likeliness of exceeding port limitations and effectively approximates II to the lower-bound value $ResMII_{mem}$. If one or more load/store nodes are not leaf nodes (i.e. the load/store node is tied by dependencies), the HLS compiler might use larger II values in order to relax the constraints generated by the lack of instruction movement. Figure 28 presents an example where a load is dependent on a store. This happens when the HLS compiler is not able to statically infer the load address, and therefore it considers that the value being requested could be the value that has just been stored. These intermediate load and store instructions cannot be easily moved, and the HLS compiler might attempt more relaxed II values.

In fact, Lin-analyzer approximates this behaviour from Vivado HLS by using an adjustment factor Iss_m. This value is calculated by finding the schedule distance (in terms of clock cycles) between two consecutive stores for array m. This value is then used to multiply the factor $\frac{Nw_m}{Pw_m}$ when calculating $ResMII_{mem}$[6].

[6] There is no Iss_m counterpart for consecutive loads, since Lin-analyzer considers that loaded values are buffered in registers, which naturally allow large fan-outs.

Figure 28 – Pipeline schedule with the presence of load/stores that cannot be easily moved. In this case, Vivado HLS reached an *II* of 6. Although this pipeline has 3 concurrent loads, they are not for the same array and thus there is no violation.



Source: Elaborated by the author.

We improve this calculation and generalise it to consider both on and off-chip memory transactions. We performed several experimental tests to better understand the Vivado HLS behaviour when scheduling reads and writes that are dependent on other memory transactions. Lina inherits the *II* approximation using *MII* from Lin-analyzer, however we provide a new calculation for $ResMII_{mem}$ as follows:

$$ResMII_{mem} = \max\{ResMII^{port}_{mem}, ResMII^{rec}_{mem}\} \tag{3.26a}$$

$$ResMII^{port}_{mem} = \max_{m}\left\{\left\lceil\frac{Nr_m}{Pr_m}\right\rceil, \left\lceil\frac{Nw_m}{Pw_m}\right\rceil\right\} \tag{3.26b}$$

$$ResMII^{rec}_{mem} = \max_{m}\{(\Delta r_m + Cr_m), (\Delta w_m + Cw_m)\} \tag{3.26c}$$

where $ResMII^{port}_{mem}$ is the *MII* value constrained by memory port, $ResMII^{rec}_{mem}$ is the *MII* counterpart constrained by dependent memory accesses, Δr_m is the largest schedule distance between dependent read transactions for memory interface m (Δw_m the counterpart for write transactions), and Cr_m is the number of connected read dependency paths for memory interface m (Cw_m is the counterpart for write transactions). The $ResMII^{port}_{mem}$ is equivalent to the $ResMII_{mem}$ calculation from Equation 3.2 and therefore Nr_m, Pr_m, Nw_m and Pw_m are also similar.

In Equation 3.2 the $ResMII_{mem}$ is calculated by finding the worst *II* constraint among all on-chip arrays m. In our updated equations, m has a more abstract meaning and it refers to memory interfaces. The exact meaning of a memory interface depends on whether the array is on or off-chip:

- Each on-chip array has their own memory interface, since each array is mapped to separate BRAM modules with dedicated read/write ports;

- For off-chip arrays, it is dependent on banking:

  – If banking is disabled, all off-chip arrays share the same read/write port and thus they share the same memory interface m;

  – If banking is enabled, each off-chip array has its own read/write interface m.

When calculating the Nr_m and Nw_m for off-chip interfaces, data packing must be taken into consideration. For example, consider that there are 100 sequential reads from interface m within a loop iteration (i.e. $Nr_m = 100$). If data packing of 4 values is possible, this value reduces to $Nr_m = \frac{100}{4} = 25$. Lina performs this calculation by counting the effective reads and writes for each interface, while ignoring DDDG nodes that have been optimised away due to data packing or redundancy.

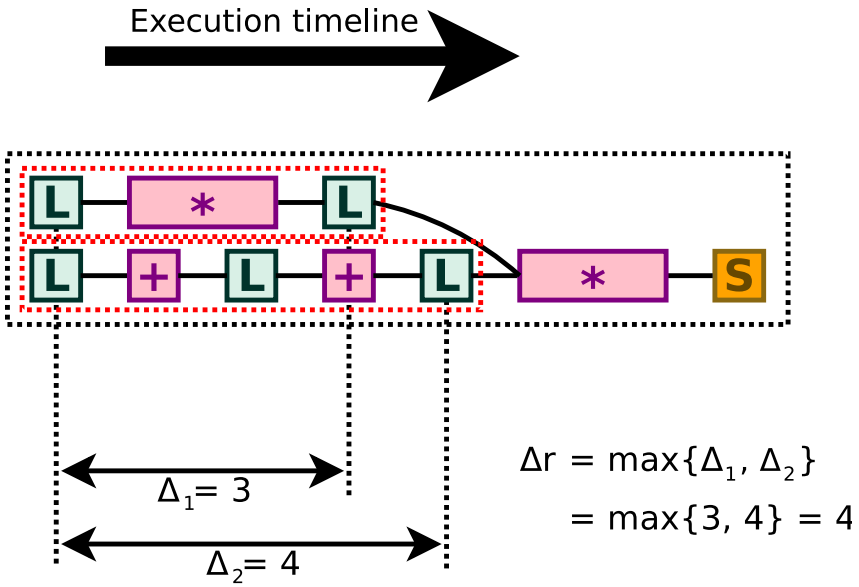
To calculate Δr_m, Lina propagates a dependency list when scheduling each node in the ASAP step. Every time a load node is scheduled, this node is added to the dependency list and propagated to children nodes. During scheduling of a new load node, if a load node is found inside the dependency list for the same memory interface m, it means that this new load is dependent on the previously scheduled load. Lina uses these lists to construct all dependency paths between reads from interface m. Then, the schedule distance between the earliest and latest nodes from each connected path is calculated, and the largest value is used as Δr_m. Similar rationale is used to calculate the write counterpart Δw_m. Figure 29 presents an example of calculating the delta values.

However, using only the delta values might still lead to invalid pipeline configurations. Considering the schedule from Figure 29, a read port conflict happens if the II is set to $\Delta r_m = 4$. The II must be relaxed to overcome this issue, and Lina approximates this behaviour by increasing the II according to the number of connected read dependency paths for the same interface (Cr_m). In Figure 29 there are two separate read dependency paths for the same interface and therefore $Cr_m = 2$. Figure 30 presents the two pipeline attempts using $II = \Delta r_m$ and $II = \Delta r_m + Cr_m$.

Similar to Lin-analyzer, we assume that the memory recurrence constraint does not affect on-chip read transactions. Therefore for on-chip arrays, the $ResMII^{rec}_{mem}$ can be simplified to:

$$ResMII^{rec}_{mem} = \max_{m}\{(\Delta w_m + Cw_m)\} \tag{3.27}$$

Figure 29 – Example of a schedule with multiple dependent loads and the respective Δr_m calculation. Assume that all loads are for the same dual-ported interface. There are two independent read dependency paths for this interface, each indicated with a red dotted box.



Source: Elaborated by the author.

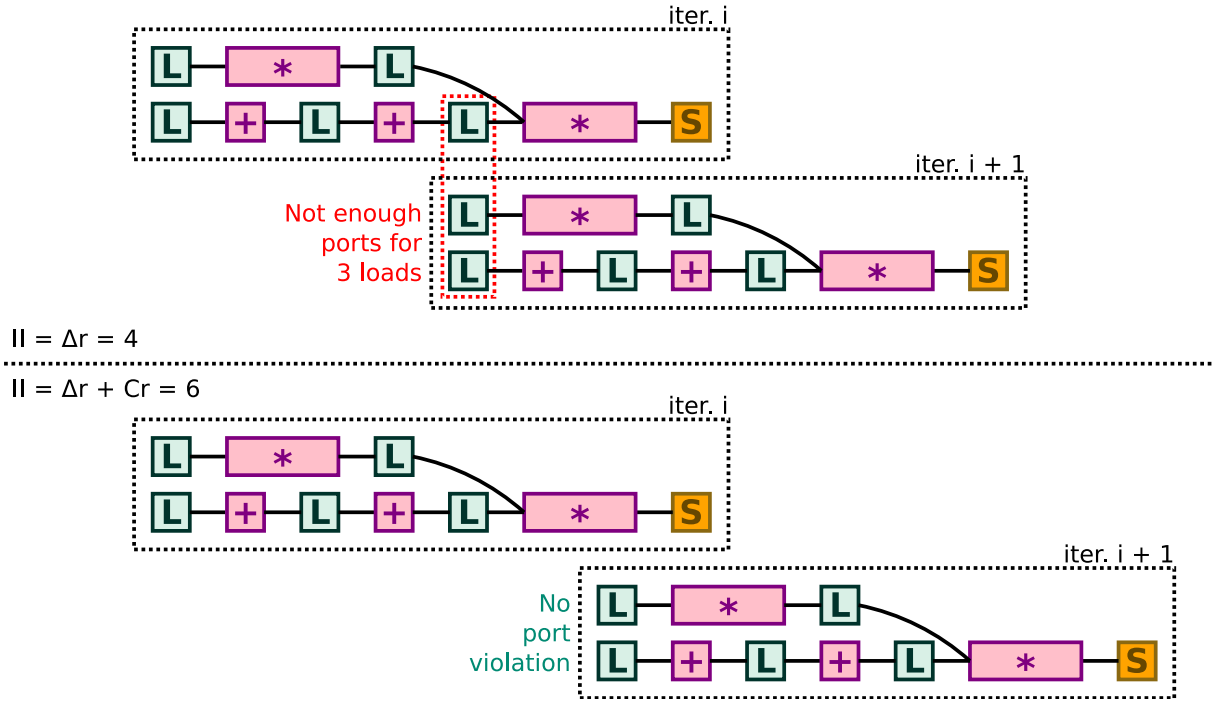
3.2.4.4 *Memory Analysis Report*

Rewriting parts of the input kernel might be required in order to satisfy the off-chip optimisations found by Lina. Currently these must be performed manually or at most semi-automatic (for example the explicit unrolling tools used in subsection 3.3.4 and section E.1). In order to reduce the burden on the high-level developer, Lina provides a memory analsysis report that indicates the optimisations found or the ones that were blocked due to constraints.

Figure 31 presents snippets of a memory report generated by Lina. The following items are reportable:

- **Informatives:** successful attempts of intra-iteration bursts, inter-iteration bursts, or data packing;
- **Warnings:** failed attempts of intra-iteration bursts, inter-iteration bursts, or data packing. Also code patterns that severely degrade performance with Vivado HLS are reported (e.g. “read-after-write” cases).

Figure 30 – Two pipeline schedules for the same computation. At top, II is set to $\Delta r_m = 4$ which triggers a read interface violation (indicated in red). At bottom, the II value is relaxed to $\Delta r_m + Cr_m = 4 + 2 = 6$. In this case, there is no port violation.



Source: Elaborated by the author.

Figure 31 – Snippets of a memory report generated by Lina.

```
[WARN] Burst possibility between loop iterations failed
       for array A
       at loop level 1
       at region before the loop nest
       Reason: detected reads for the same array
               at loop level 2
               within the loop nest

...

[WARN] Vectorisation attempt failed
       for array B
       with 4 elements vectorised
       Reason: cannot align write with pack size
               due to 3 unused element(s) after
               at loop level 1
               at region before the loop nest

...

[INFO] Vectorisation attempt successful
       for array C
       with 4 elements vectorised
```

Source: Elaborated by the author.

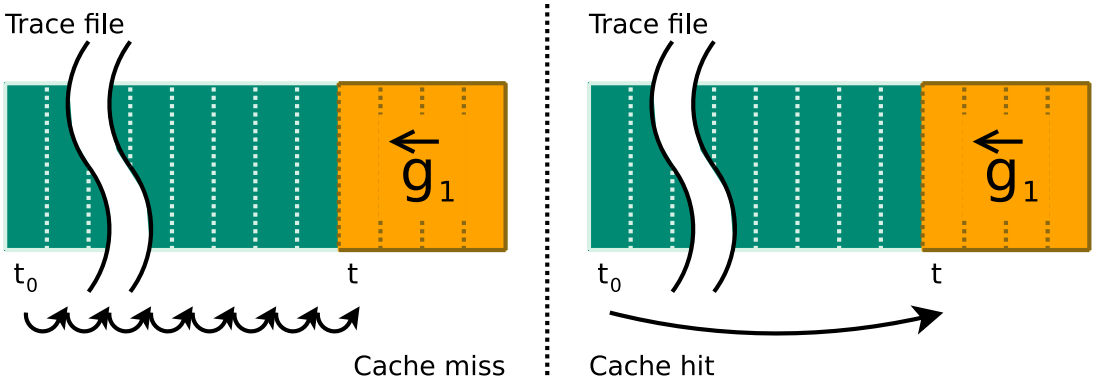
3.2.5 DSE Temporal Locality Caching

The dynamic traces generated from profiling can reach large sizes, even for loop nests with moderate dimensions. Since the trace is linearly parsed instruction by instruction, it can take a significant time to locate and parse the instructions to generate multiple DDDGs if they are not close to each other in the trace.

During the generation of a DDDG, consider that the trace cursor (i.e. the current file position of the trace) is t_0. We noticed that starting from t_0, Lina always traverses to the same trace cursor t when generating a specific type of DDDG g, regardless of which optimisations are enabled or disabled for the current design point. This indicates that there is a temporal locality between the evaluation of different design points.

We implement a trace cursor cache in order to avoid traversing the trace file if the position to generate a certain type of DDDG is known from previous design points. For example, consider that the generation of a `before` DDDG at loop level 1 ($\overleftarrow{g}_1$) is requested and that the current trace cursor is t_0. If the cache is empty, Lina performs the trace traverse as usual until position t is reached, where the first instruction that should be part of $\overleftarrow{g}_1$ is found. Then, t is stored in the cache using $(t_0, \overleftarrow{g}_1)$ as a key. If during the evaluation of another design point the same condition is met (i.e. $\overleftarrow{g}_1$ must be generated and the current trace cursor is t_0), Lina avoids the traversal and sets the trace cursor to t as informed by the cache. This example is depicted in Figure 32.

Figure 32 – Two examples of dynamic trace traversal for the generation of DDDG $\overleftarrow{g}_1$. At left, there is a cache miss, and the trace must be traversed instruction-wise until the first instruction from $\overleftarrow{g}_1$. At right, there is a cache hit and Lina can proceed directly to the cached cursor t.



Source: Elaborated by the author.

Each of the p threads in our parallel job dispatcher uses its own cache file since we do not define any race condition logic.

3.2.6 Exploration Quality Metrics

After the exploration, we define a Pareto frontier P_{lin} by minimising resource usage and execution time based on the previously obtained estimations. We compare P_{lin} to the golden Pareto frontier P_{viv} generated by running Vivado for all design points.

Consider that D is the set of all enumerated design points $\mathbf{x}$ for exploration, O is the set of objectives i (e.g. execution time, LUTs, FFs) and $f_i(\mathbf{x})$ is the true value of objective i associated with design point $\mathbf{x}$. We compare P_{lin} against P_{viv} using the Average Distance from Reference Set (ADRS) (CZYŻAK; JASZKIEWICZ, 1998). The ADRS calculation couples each true Pareto point $\mathbf{y} \in P_{\text{viv}}$ to the closest estimated point $\mathbf{x} \in P_{\text{lin}}$ using an error metric between the objectives $f_i(\mathbf{y})$ and $f_i(\mathbf{x})$ as proximity criteria. Then, the worst objective deviation of each pair $(\mathbf{y}, \mathbf{x})$ is accumulated and averaged. Low ADRS values indicate that each $\mathbf{y}$ has a close estimated design point $\mathbf{x}$ and the value itself indicates the average worst objective deviation. The ADRS is defined as:

$$ADRS = \frac{1}{|P_{\text{viv}}|} \cdot \sum_{\mathbf{y} \in P_{\text{viv}}} \{ \min_{\mathbf{x} \in P_{\text{lin}}} \{c(\mathbf{x}, \mathbf{y})\} \} \tag{3.28a}$$

$$c(\mathbf{x}, \mathbf{y}) = \max_{i \in O} \{0, e_i(\mathbf{x}, \mathbf{y})\} \tag{3.28b}$$

where $c(\mathbf{x}, \mathbf{y})$ is an achievement-scalarising function that calculates the worst objective deviation e_i between design points $\mathbf{x}$ and $\mathbf{y}$.

This metric is in the form of:

$$e_i(\mathbf{x}, \mathbf{y}) = \frac{f_i(\mathbf{x}) - f_i(\mathbf{y})}{\Delta_i} \tag{3.29}$$

where Δ_i is a normalisation factor that adjusts all objectives to be on a similar scale. This is needed since different objectives are quantitatively compared by the ADRS calculation.

We use two options for $e_i(\cdot)$. First, we consider the distance relative to the true objective value (PALERMO; SILVANO; ZACCARIA, 2009; ZHONG *et al.*, 2014):

$$e_i(\mathbf{x}, \mathbf{y}) = \frac{f_i(\mathbf{x}) - f_i(\mathbf{y})}{f_i(\mathbf{y})} \tag{3.30}$$

The other formulation for e_i that we consider uses the distance relative to the Pareto range (CZYŻAK; JASZKIEWICZ, 1998):

$$e_i(\mathbf{x}, \mathbf{y}) = \frac{f_i(\mathbf{x}) - f_i(\mathbf{y})}{f_i(\mathbf{y}_{\max}) - f_i(\mathbf{y}_{\min})} \tag{3.31}$$

where $\mathbf{y}_{max}$ and $\mathbf{y}_{min}$ are the design points that provide the maximum and minimum values for the objective i in P_{viv}. We refer to the ADRS using Equation 3.30 as $ADRS_{rel}$ and the one using Equation 3.31 as $ADRS_{par}$.

We calculate both values in our validation since they indicate different characteristics. The $ADRS_{rel}$ is useful for quantifying how much worse our estimations are compared to the true optimal points. However, small values of $\Delta_i = f_i(\mathbf{y})$ may excessively accentuate the error even if both $f_i(\mathbf{x})$ and $f_i(\mathbf{y})$ are small in comparison to the total budget available of objective i. On the other hand, $ADRS_{par}$ considers the Pareto interval. In this case, the differences between $f_i(\mathbf{x})$ and $f_i(\mathbf{y})$ are significantly highlighted when the interval is small.

However, there is no guarantee that every design point in D will evenly represent the intervals of each objective. Thus, a high value of ADRS does not necessarily imply that many other points could result in better approximations than the ones given by Lina.

Let P_{pair} be the set of pairs $(\mathbf{y}, \mathbf{x})$, where each true Pareto point $\mathbf{y} \in P_{viv}$ is associated with its closest estimated Pareto point $\mathbf{x} \in P_{lin}$. Let $Q_{\mathbf{x},\mathbf{y}}$ be the set of all non-Pareto points $\mathbf{z}$ that are better than the estimated Pareto point $\mathbf{x}$ in respect to $\mathbf{y}$, as follows:

$$Q_{\mathbf{x},\mathbf{y}} = \{\mathbf{z} \in D - P_{viv} | \max\{0, e_i(\mathbf{z},\mathbf{y})\} \leq \max\{0, e_i(\mathbf{x},\mathbf{y})\}, \forall i \in O\} \tag{3.32}$$

We propose a complementary metric named Near-Optimal Density (NOD) that provides an insight of the proportion of design points that could be better than those selected by the DSE with Lina. NOD is defined as:

$$NOD = \frac{1}{|P_{viv}|} \cdot \sum_{(\mathbf{y},\mathbf{x}) \in P_{pair}} \left\{ \frac{|Q_{\mathbf{x},\mathbf{y}}|}{|D|} \right\} \tag{3.33}$$

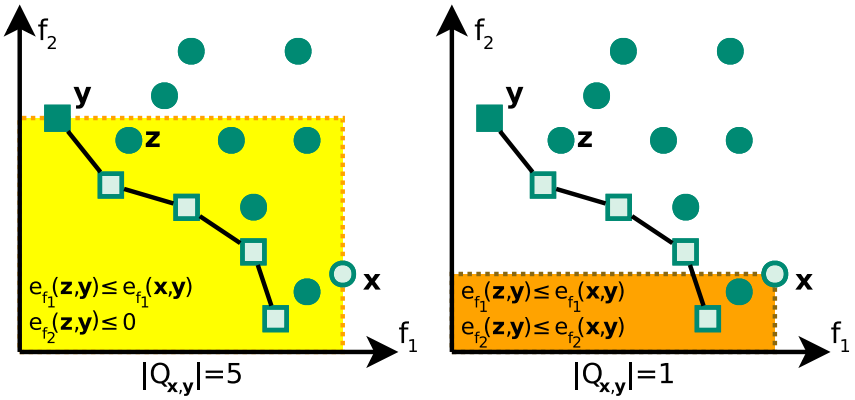
Note that we use a max operator in Equation 3.32 to replace negative e_i values by 0, similar to the ADRS calculation. Without this approach, a Pareto estimation containing points that are very optimised for few — but not all — objectives could unfairly reduce the NOD value by excluding other points that are closer to the true Pareto points. An example of this scenario is shown in Figure 33.

3.3 Experimental Setup

This section presents the evaluation setup created to both validate our DSE approach and to compare it against related work, the platforms used, and the CNN experiments that we performed to evaluate the off-chip memory model.

We perform three validations. First, we compare Lina against Lin-analyzer by performing a small exploration without optimisation of resources, as better described in subsection 3.3.2. Then, we perform a larger exploration aiming the optimisation of design

Figure 33 – Example of a two-objective design space, where the true Pareto points are indicated as square points, the estimated Pareto point as $\mathbf{x}$, and the error metrics as e_{f_1} and e_{f_2}. The coloured regions represent the points $\mathbf{z} \in D - P_{\text{viv}}$ that compose $Q_{\mathbf{x},\mathbf{y}}$. In this example, $e_{f_2}(\mathbf{x},\mathbf{y})$ is negative. At left, $Q_{\mathbf{x},\mathbf{y}}$ is constructed according to Equation 3.32. At right, the predicate that defines $Q_{\mathbf{x},\mathbf{y}}$ does not use the max operator. Many other points $\mathbf{z}$ are better approximations to $\mathbf{y}$ than $\mathbf{x}$ when considering both objectives, even though $\mathbf{x}$ has a smaller f_2 objective than $\mathbf{y}$. One example of such point $\mathbf{z}$ is shown in the figure. The left case better reflects this scenario by including more $\mathbf{z}$ points in $Q_{\mathbf{x},\mathbf{y}}$, leading to a larger NOD value.



Source: Perina *et al.* (2021).

latency and resources, as described in subsection 3.3.3. Finally, we perform a validation considering off-chip memory accesses using a convolutional kernel, as shown in subsection 3.3.4.

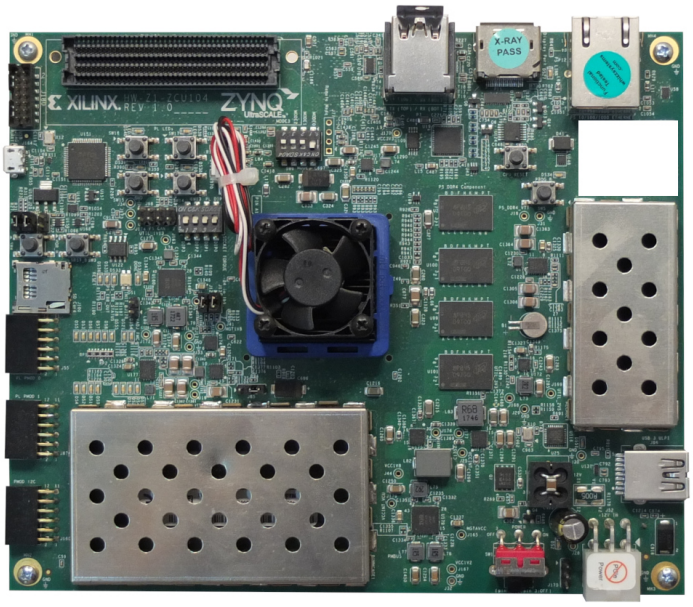
3.3.1 Platforms and Software Used

Our target FPGA platform is the Xilinx Zynq UltraScale+, depicted in Figure 34. The Zynq UltraScale+ is a SoC that contains an ARM processor and an FPGA. By using the SDSoC toolchain supplied by Xilinx, one is able to execute a host C/C++ code on the ARM side, while deploying certain functions to be executed on the FPGA side using HLS. We use the version 2018.2 of the toolchain in this thesis. The syntheses of all design points were performed in a system with an AMD EPYC 7702P CPU, and the DSE with Lina was performed in a system with an Intel i7-5500U CPU.

For the first validation (subsection 3.3.2), the ZCU102 model of the FPGA platform was used. For the remaining validations, we used a ZCU104 platform. Both models are from the same family and thus the FU characterisation is the same for both. The only difference is related to the amount of resources available. In order to provide a fair comparison against Lin-analyzer in the first experiment, we adapted Lin-analyzer to support the ZCU102 platform as well.

Kernels targetting SDSoC can be written either in C/C++ or OpenCL. For the experiments that do not involve off-chip memory accesses, we use C/C+ as primary

Figure 34 – Xilinx Zynq UltraScale+ ZCU104 development kit.



Source: Elaborated by the author.

input and almost no effort is required on converting the pure software codes to SDSoC C/C++ projects. For the convolution experiments that involve off-chip transactions, we use OpenCL instead due to two reasons. First, OpenCL provides by specification a memory hierarchy that involves off-chip memories, therefore off-chip memory access with OpenCL is possible simply by accessing arrays marked with the `__global` keyword. Second, OpenCL supports vector sizes of common data types (e.g. `float4` for 4 packed floats in a single word), which is very suitable for the data packing analysis performed by Lina. Note that we do not convert the input C/C++ functions to the OpenCL's SIMD (i.e. NDRange) model, therefore migrating from one language to another does not require much effort. It can be as simple as changing the function header from C to OpenCL standard and adding OpenCL API calls to the host code. Apart to some minor keyword adaptations, the function body is left practically intact.

3.3.2 First Validation: Comparison Against Lin-Analyzer

In this first validation, our intention is to assess the improvements brought by the timing-constrained scheduler and non-perfect loop analyzer when compared to Lin-analyzer. We used the same kernels as supplied in the Lin-analyzer's repository, as presented in Chart 4.

We performed a small parameters exploration (less than 100 points) with loop unroll, pipelining, array partitioning and clock frequency, as shown in Chart 5. Each configuration is identified by an ID, which is formed by concatenating the values from the

Chart 4 – Kernels used in the first validation.

Kernel	Description	Dimensions
`atax`	Matrix transpose and vector mult.	128, 128
`bicg`	Biconjugate gradients	256, 256
`conv2d`	2D convolution	126, 126
`conv3d`	3D convolution	30, 30, 30
`gemm`	Matrix multiply	128, 128, 128
`gesummv`	Scalar, vector and matrix multiply	128
`mvt`	Matrix vector product and transpose	256
`syr2k`	Symmetric rank-2k operations	128, 128
`syrk`	Symmetric rank-k operations	128, 128

Source: Elaborated by the author.

table in a bitwise manner. For example, a configuration with partitioning factor 4 (option 10), 114.29MHz[7] (option 1), no pipelining (option 0) and unroll at the two innermost loops (option 11) has a binary ID of 101011 or decimal ID 43. For each configuration, we measured the relative percentage error from Lina and Lin-analyzer estimations against the values reported by Vivado HLS.

Chart 5 – Parameters used in the exploration.

Description	Possible values
Array partitioning	00: No partitioning 01: Some arrays complete, some with factor of 2 10: Some arrays complete, some with factor of 4 11: Some arrays complete, some with factor of 8
Effective freq.	0: 136.98MHz (10ns period, 27% uncertainty) 1: 114.29MHz (10ns period, 12.5% uncertainty)
Loop pipelining	0: No pipeline 1: Pipeline at innermost level
Loop unroll	00: No unroll 01: Unroll at innermost level 10: Unroll at 2nd-innermost level 11: Unroll at innermost and 2nd-innermost level

Source: Elaborated by the author.

In addition to comparing against Lin-analyzer, we also compare our results against COMBA[8] for two kernels: `bicg` and `gemm`. COMBA performs the DSE in a single execution,

[7] These frequency values seems counter-intuitively picked, however they are resultant from different uncertainties being applied to an 100MHz frequency. With 27% uncertainty, the frequency considered by Vivado HLS raises to 136.98MHz. With 12.5%, the value raises to 114.29MHz.

[8] Code available at <https://github.com/zjru/COMBA>.

and the fastest design point is given at the end.

3.3.3 Second Validation: Resource and Timing-aware Exploration

The second validation set is composed of 11 floating-point kernels, and 5 kernels with integer and bitwise operations. The floating-point kernels are from the PolyBench benchmark, and the non-floating-point kernels are from an AES security module for FPGAs that provides data and design confidentiality (SILITONGA *et al.*, 2018). The AES module is part of a support system that does not compose the main application to be mapped on the FPGA. Therefore, it is a suitable use case for our exploration since both performance and resource footprint are important objectives to be optimised. Additionally, our exploration is suitable for the AES kernels since Lina supports optimising resources on kernels that do not have any floating-point calculation.

In order to obtain P_{viv}, each design point must be synthesised. For design spaces with thousands of design points, the total evaluation of all combinations becomes impractical. Therefore we define two different experiments — `hls` and `fullsyn` — that use reports from different stages of the Vivado synthesis. The `hls` uses the early resource estimation reports from Vivado HLS, allowing to explore from hundreds to few thousands of design points. The `fullsyn` uses the final resource usage reports from SDSoC, which are produced only after the time demanding hardware synthesis process is completed. As a result, the design spaces are reduced to less than 150 design points. Chart 6 presents the 16 kernels that compose our validation set along with a brief description, the data dimensions for the PolyBench kernels, and the number of valid design points in the experiments. The dimensions were used to define the array sizes and loop bounds.

Some of the kernels present in this validation are the same as ones used in the previous validation (i.e. Chart 4). We removed the ones that were not part of the official PolyBench repository (e.g. the `conv2d` and `conv3d` kernels) and also ones that had a similar computation pattern to other kernels already present (e.g. `atax`). In this validation, we use more aggressive compilation pragmas than the ones in the previous validation (e.g. larger unroll factors, or pipeline directives in other loops than the innermost). For this reason, some kernels had their dimensions adjusted to avoid cases that are too complex to fit in the platforms used.

While preserving functional behaviour, some kernels were modified to make them compatible with Lina limitations (see section 3.5 for more information about the limitations). A more detailed description of these modifications — including the kernels itself — can be found in the project repository.

For each kernel, we selected loop unroll factors that divide the respective loop bounds with no remainder. The same is applied for array partition factors and respective

Chart 6 – Validation kernel set.

PolyBench (floating-point)				
			# Points	
Name	Description	Dim.	hls	fullsyn
`bicg`	Biconjugate gradients	390, 410	2304	64
`floyd`	Floyd-Warshall	60	2430	96
`gemm`	Matrix multiply	60, 70, 80	5832	128
`gesummv`	Scalar-vec.-matrix mul.	250	1536	96
`heat3d`	3D heat equation	50	3240	72
`jacobi1d`	1D Jacobi stencil	5000	540	96
`jacobi2d`	2D Jacobi stencil	302	1536	96
`mvt`	Matrix vec. prod. transp.	400	3456	96
`seidel2d`	2D Seidel stencil	122	480	72
`syr2k`	Symmetric rank-2k	60, 80	1728	128
`syrk`	Symmetric rank-k	60, 80	972	96
AES (int/byte ops)				
			# Points	
Name	Description		hls	fullsyn
`KeyExp`	Key schedule		320	96
`KeyExp2`	Key sched. (SBOX is partitioned)		1280	128
`MixCols`	Column mix		10176	80
`SubBytes`	Byte substitution		1008	96
`SubShfMix`	Byte substitution, shift and mix		320	144

Source: Perina *et al.* (2021).

array sizes. Then, we limited the number of combinations for each optimisation knob in order to have feasible design space sizes. Chart 7 presents the optimisation knobs used in both experiments. We use different clock frequencies with `fullsyn`, since SDSoC only allows the selection of a limited amount of frequencies.

3.3.3.1 Additional Experiments on Non-perfect Kernels with Larger Loop Bounds

To assess the impact of the non-perfect loop analysis, we considered the three kernels from PolyBench that contain non-perfect loop nests: `gemm`, `syr2k` and `syrk`. However, their non-perfect segments are not significantly larger than the innermost loop. Therefore, to better emphasise the impacts of our analysis, we modified some constants to increase the execution fraction of the non-perfect segments:

- `gemm`: constant `NJ` increased from 70 to 700;
- `syr2k`: constant `N` increased from 80 to 320;

Chart 7 – Optimisation knobs.

Type	`hls`	`fullsyn`
Period (ns)	5, 7.5, 10, 15, 17.5, and 20	5, 6.66, 10, and 13.33
Loop unroll	Applicable to all loop levels	Applicable mostly to the inner loop levels
Loop pipeline	Applicable mostly to the inner loop levels	Applicable to the innermost loop level (or at most its parent)
Array partition	Cyclic and block possible, complete only for small arrays	Same as `hls`, but fewer options

Source: Perina *et al.* (2021).

- `syrk`: constant `N` increased from 80 to 320.

These constants are used to define the array sizes and loop bounds. No modifications were made to the exploration knobs.

3.3.3.2 Comparison with Related Work

Similar to the previous validation, we use COMBA for comparison against our DSE approach. COMBA's resource estimation is only used to check the feasibility of each point given a constrained budget of BRAMs and DSPs[9]. Moreover, the clock frequency is fixed for each exploration and must be defined at compile-time. Using our validation set, we create four comparison experiments with different frequencies and resource budgets: `rtot`, `r50`, `r10` (all at 100MHz), and `rtot200` (at 200MHz). The whole DSP and BRAM resource budget of COMBA's target FPGA (Xilinx Virtex-7) is available in experiments `rtot` and `rtot200`, whereas in `r50` and `r10` we restrict to 50% and 10% of the budget, respectively.

For these experiments, we synthesise the design points given by our DSE for the Virtex-7 platform, the same used by COMBA. Even though our exploration is fine-tuned for the ZCU104 platform, we believe that the comparison is still valid since both FPGAs are from the same vendor and use the same HLS tool for the compilation process.

3.3.4 Third Validation: Off-chip Experiments in the CNN Context

Convolutional neural networks are particular applications that have a code structure suitable to parallelism, and also a large memory footprint for its inputs and outputs.

[9] The COMBA code openly available calculates LUTs, but does not constrain them.

There are several studies focused on using smaller on-chip buffers and loop reordering/tiling (STOUTCHININ; CONTI; BENINI, 2019; ZHANG *et al.*, 2015; PEEMEN *et al.*, 2013). Their primary focus is to optimise the access to off-chip memory while maximising the usage of on-chip resources.

Source code 4 presents the basic loop nest of a CNN kernel. Its code pattern is suitable for memory optimisation, and therefore we validate our memory model using Lina to explore a single CNN layer. These experiments are focused on performance, and therefore we do not perform any resource-aware or Pareto analysis. We use the configuration of ZFNet's 6th layer as in Stoutchinin, Conti and Benini (2019), presented in Chart 8.

Source code 4 – Basic loop nest of a CNN kernel

```
1: LOF: for(auto m = 0; m < M; m++)
2:   LIF: for(auto c = 0; c < C; c++)
3:     LSY: for(auto y = 0; y < E; y++)
4:       LSX: for(auto x = 0; x < E; x++)
5:         LFY: for(auto k = 0; k < E; k++)
6:           LFX: for(auto l = 0; l < E; l++) {
7:             auto p = I[c][y * S + k][x * S + l];
8:             auto w = W[m][c][k][l];
9:             O[m][y][x] += p * w;
10:          }
```

Chart 8 – ZFNet CNN layer configuration used.

Name	Description	Value
C	# input feature maps	256
M	# output feature maps	256
H	Input feature map size (H × H)	6
R	Convolution kernel size (R × R)	3
S	Convolution kernel stride	1
E	Output feature map size (E × E)	6

Source: Elaborated by the author.

We explore two small variations of a CNN layer that differs on how the read/write border cases are handled. The first version, named `padmemory`, has the padding elements embedded on the input and output arrays. This means that the kernel code is similar to the Source code 4, with no special treatment for border cases. In the second version, named `padlogic`, the arrays have no embedded padding and additional `if-else` blocks are required to control the read/write indexes and avoid out-of-bounds memory accesses. Source code 5 presents the `padlogic` kernel.

Source code 5 – The padlogic CNN kernel

```
1: LOF: for(auto m = 0; m < M; m++)
2:   LIF: for(auto c = 0; c < C; c++)
3:     LSY: for(auto y = 0; y < E; y++)
4:       LSX: for(auto x = 0; x < E; x++)
5:         LFY: for(auto k = 0; k < E; k++)
6:           LFX: for(auto l = 0; l < E; l++) {
7:             auto h1 = y * S + k, h2 = x * S + l;
8:             auto p = (h1 < 0 || h1 >= H || h2 < 0 || h2 >= H)? 0 :
9:               I[c][y * S + k][x * S + l];
10:            auto w = W[m][c][k][l];
11:            O[m][y][x] += p * w;
12:          }
```

The input arrays `I` and `W` are left off-chip, while the output array `O` is buffered on-chip during execution. The results from `O` must be transferred back to the off-chip memory after the CNN layer completes. We leave memory banking always enabled, as there is no counter indication for not using it. Chart 9 presents the design space knobs that we used to explore both `padmemory` and `padlogic` variants. Both design spaces have 680 valid points.

Chart 9 – Optimisation knobs for the CNN kernels.

Loop knobs			
Name	Loop depth	Unroll factors	Pipeline
LOF	1	Off	Off
LIF	2	Off	Off
LSY	3	Off, 2	Off, on
LSX	4	Off, 2	Off, on
LFY	5	Off, 3	Off, on
LFX	6	Off, 3	Off, on
Array knobs			
Name	I/O	Partitioning	
I	Input	No partitioning (off-chip access)	
W	Input	No partitioning (off-chip access)	
O	Output	**Off**; **Block:** 4, 8; **Cyclic:** 4, 8	
Frequencies			
Values (MHz)		75, 100, 150, 200	

Source: Elaborated by the author.

As explained in paragraph 3.2.4.3.1, using SDSoC unroll pragmas may generate

"read-after-write" patterns that block the `permissive` policy. Thus, we created an automated tool to explicitly unroll the CNN layer and place all read values before writes when generating the HLS code. Source code 6 presents an example of CNN layer code generated by this tool, with the inner loop unrolled with a factor of 4.

Source code 6 – Explicitly unrolled padmemory kernel, with all reads placed before writes

```
1: LOF: for(auto m = 0; m < M; m++)
2:   LIF: for(auto c = 0; c < C; c++)
3:     LSY: for(auto y = 0; y < E; y++)
4:       LSX: for(auto x = 0; x < E; x++)
5:         LFY: for(auto k = 0; k < E; k++)
6:           LFX: for(auto l = 0; l < E; l += 4) {
7:             auto __p_0 = I[c][y * S + k][x * S + l];
8:             auto __w_0 = W[m][c][k][l];
9:             auto __p_1 = I[c][y * S + k][x * S + l + 1];
10:            auto __w_1 = W[m][c][k][l + 1];
11:            auto __p_2 = I[c][y * S + k][x * S + l + 2];
12:            auto __w_2 = W[m][c][k][l + 2];
13:            auto __p_3 = I[c][y * S + k][x * S + l + 3];
14:            auto __w_3 = W[m][c][k][l + 3];
15:
16:            O[m][y][x] += __p_0 * __w_0;
17:            O[m][y][x] += __p_1 * __w_1;
18:            O[m][y][x] += __p_2 * __w_2;
19:            O[m][y][x] += __p_3 * __w_3;
20:          }
```

For each kernel, we perform four experiments by toggling the memory policies and the data packing directive. Each toggle is represented by a pair of keywords `novec`/`vec`, `cons`/`perm` and `unrviv`/`unrexp`. Chart 10 describes these keywords and how they relate to each experiment.

3.4 Results

In the following subsections, we present the results for the experiments described in the previous section, namely the comparison against Lin-analyzer, the resource and timing-aware exploration — both with no off-chip accesses — and finally the off-chip explorations with the CNN kernels.

Chart 10 – Convolution experiments.

Exp.	Aspects	
#	Name	Description
1	`novec`	Lina DSE is performed without data packing analysis
	`cons`	Lina DSE is performed using the `conservative` memory policy
	`unrviv`	The HLS code is unrolled using SDSoC directives
2	`vec`	Lina DSE is performed with data packing analysis
	`cons`	Lina DSE is performed using the `conservative` memory policy
	`unrviv`	The HLS code is unrolled using SDSoC directives
3	`novec`	Lina DSE is performed without data packing analysis
	`perm`	Lina DSE is performed using the `permissive` memory policy
	`unrexp`	The HLS code is explicitly unrolled using the tool described above
4	`vec`	Lina DSE is performed with data packing analysis
	`perm`	Lina DSE is performed using the `permissive` memory policy
	`unrexp`	The HLS code is explicitly unrolled using the tool described above

Source: Elaborated by the author.

3.4.1 First Validation: Comparison Against Lin-Analyzer

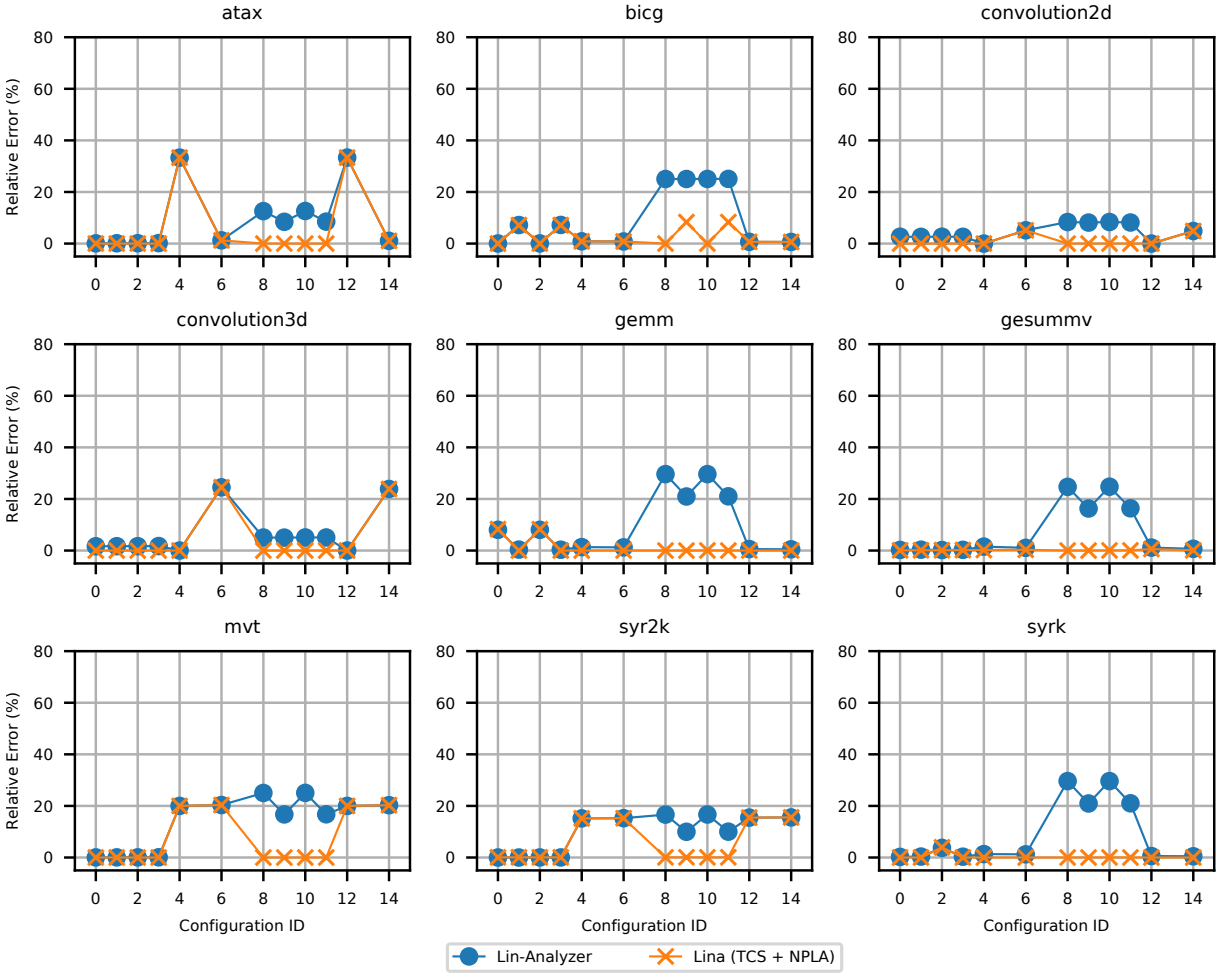
Figure 35 and Figure 36 present Lina and Lin-analyzer's estimation error relative to the actual values from Vivado HLS, without and with array partitioning respectively. To avoid polluted plots, we show only partitioning factor of 2 (IDs 16-30). The complete spreadsheets can be found in Lina's repository.

Both estimators presented little or no error without partitioning and pipelining (ID $<$ 4) and Lina had an improved accuracy on `conv2d` and `conv3d`. With pipelining (IDs 4 and 6) both had comparable results, with some configurations with error of $\sim$ 20%. Until this point both estimators were configured to the same target frequency (100MHz with 27% uncertainty). For the IDs 8-14, a different frequency was used and Lina gave in many cases the exact cycle count, while Lin-analyzer deviated from 10% to 30%. One source for the mentioned errors is due to internal optimisations performed by Vivado that both estimators do not reproduce.

With array partitioning enabled (ID $\geq$ 16), Lina presented improved overall accuracy, but for some configurations both had notable deviations, with a peak value of $\sim$ 65%. One of the reasons is that Vivado conservatively assumes loop-carried dependencies that Lina and Lin-analyzer do not. In some cases, Lin-analyzer performed better than Lina. This is due to some internal differences between both estimators, for example in the σ value used on Equation 3.6. Lina considers $\sigma = 2$, whereas Lin-analyzer considers a σ of three[10]. If both Lina and Lin-analyzer overestimate the value for a design point, Lin-

[10] We adjusted the σ value of Lina from 3 to 2 after some experimentation with Vivado HLS.

Figure 35 – Lina/Lin-analyzer estimation errors relative to the cycle counts reported by Vivado HLS. The x-axis represent different pragma configurations, and partitioning is disabled.



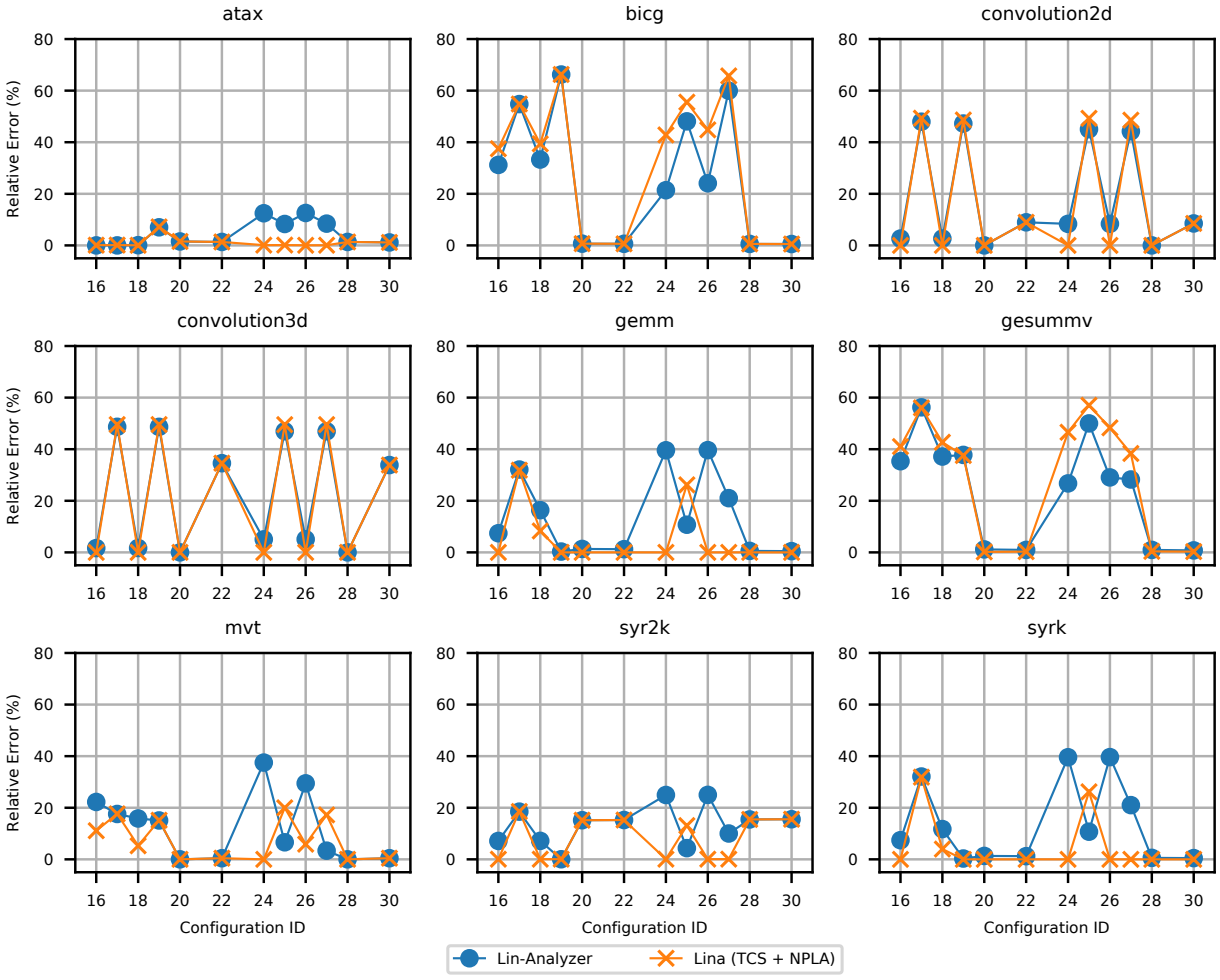
Source: Research data.

analyzer might provide a better result because of the reduced σ value, which inadvertedly reduces the estimation and in turn reduces the error.

In average for all kernels and tested configurations, the estimations from Lin-analyzer had a relative error of 16.45%, while Lina 13.01%. Excluding array partitioning, the error for Lin-analyzer dropped to 8.85% and Lina to 3.02%. Thus, in average, Lina presented better accuracy than Lin-analyzer.

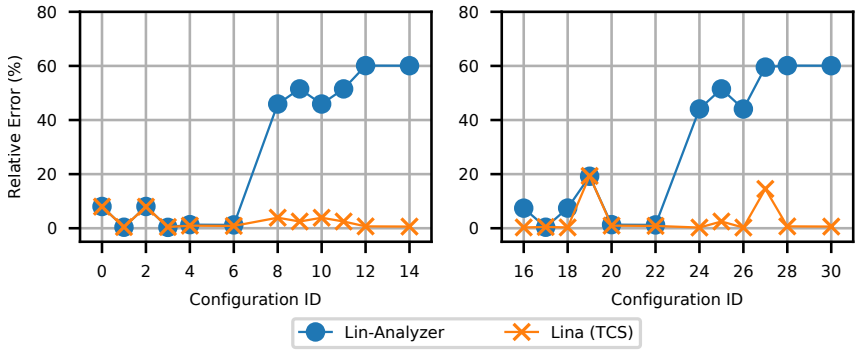
Figure 37 presents the relative errors for `gemm` with a larger frequency for IDs 8-14 and 24-30. As expected, the increased frequency disparity from the fixed 100MHz of Lin-analyzer yielded larger error for these configurations, since it does not perform any timing analysis.

Figure 36 – Lina/Lin-analyzer estimation errors relative to the cycle counts reported by Vivado HLS. The x-axis represent different pragma configurations, and partitioning is enabled.



Source: Research data.

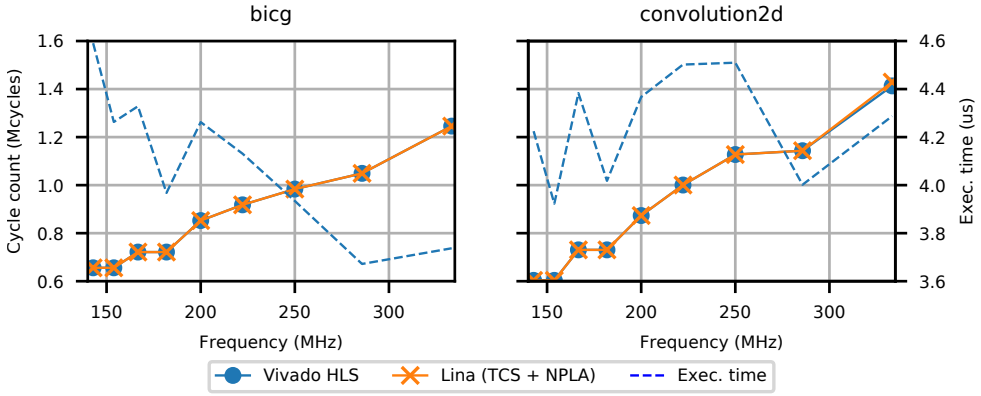
Figure 37 – Results for `gemm` with higher frequency for IDs 8-14 and 24-30.



Source: Perina, Becker and Bonato (2019a).

Figure 38 presents the cycle count for `bicg` and `conv2d` under different frequencies with optimisations disabled. It can be noted that Lina accurately reacted to the different timing constraints. Furthermore, the peak performance was not at the highest frequency nor the lowest cycle count: the optimal frequency was estimated at 285.71MHz and 153.85MHz for `bicg` and `conv2d` respectively.

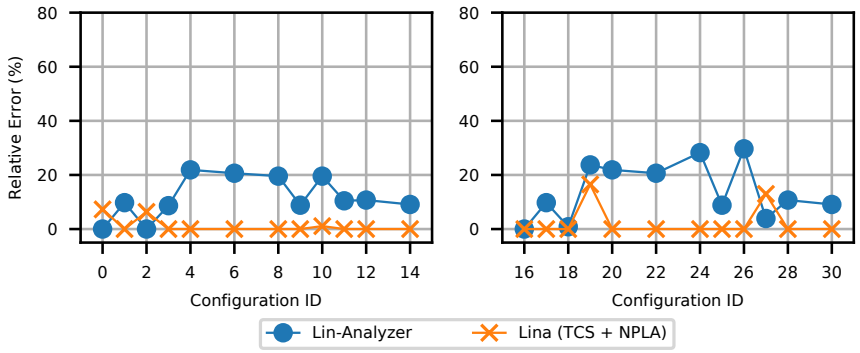
Figure 38 – Cycle count reported by Lina and Vivado HLS for `bicg` and `conv2d` in different frequencies (all optimisations disabled). The dashed line represents the design execution times considering frequency and cycle count. Both plots are scaled to the same intervals.



Source: Perina, Becker and Bonato (2019a).

Figure 39 presents a variation of `gemm` where different loop bounds were used in a way that the proportion of executed operations between the outer loops and the innermost loop increased. As expected, the error difference between Lina and Lin-analyzer was more noticeable, since Lin-analyzer ignores the instructions between loop nests. For three configurations Lina had worse accuracy, which was caused by a similar effect as the one previously explained involving the σ value.

Figure 39 – Relative errors for a variant of `gemm` with different loop bounds.



Source: Perina, Becker and Bonato (2019a).

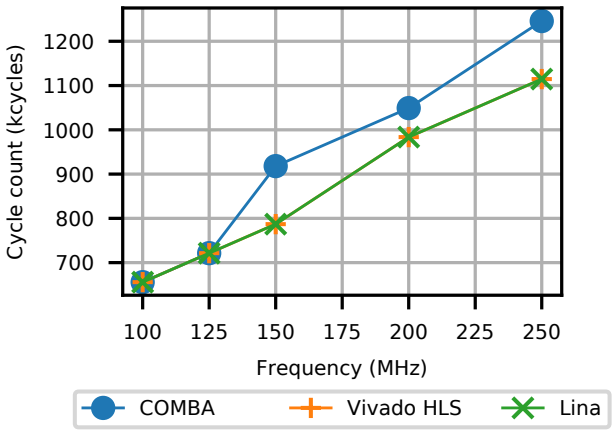
Regarding Lina's execution time, the trace part was the most significant portion of the total time, accounting for 95% and 93% of the time of Lin-analyzer and Lina respectively. Since it has to be performed only once, its impact decreases with the design space

size. For the experiments here performed, the total exploration time was 7.50 minutes with Lin-analyzer and 7.67 with Lina, while Vivado HLS took 397.91 minutes.

In comparison to COMBA, we executed both estimators for `bicg` and `gemm` kernels with large and small loop bounds. For `bicg`, COMBA explored 1758 and 514 points for large and small variants respectively, leading to a per-point estimation time of 0.40s and 0.46s against Lina's 0.12s and 0.05s including trace. For the `gemm` kernel, 1444 and 604 points were explored by COMBA for the large and small variants, leading to per-point 0.35s and 0.38s while Lina took 1.75s and 0.07s. The increased time for `gemm` large bounds with Lina is due to the dominating trace time. which better dissolves with larger design spaces (as shown in the next subsection).

Even though COMBA also performs timing analysis and cycle merging, it has only 5 options of frequencies. We created a preliminary hardware profile for the board used in their paper and compared the cycle count from `bicg` against the actual values of Vivado for different frequencies as shown in Figure 40. It can be noted that COMBA presents deviation from 150MHz onwards, while Lina keeps its accuracy.

Figure 40 – Cycle count comparison between COMBA, Lina and Vivado HLS for the `bicg` kernel at different frequencies (all optimisations disabled), using a preliminary hardware profile library.

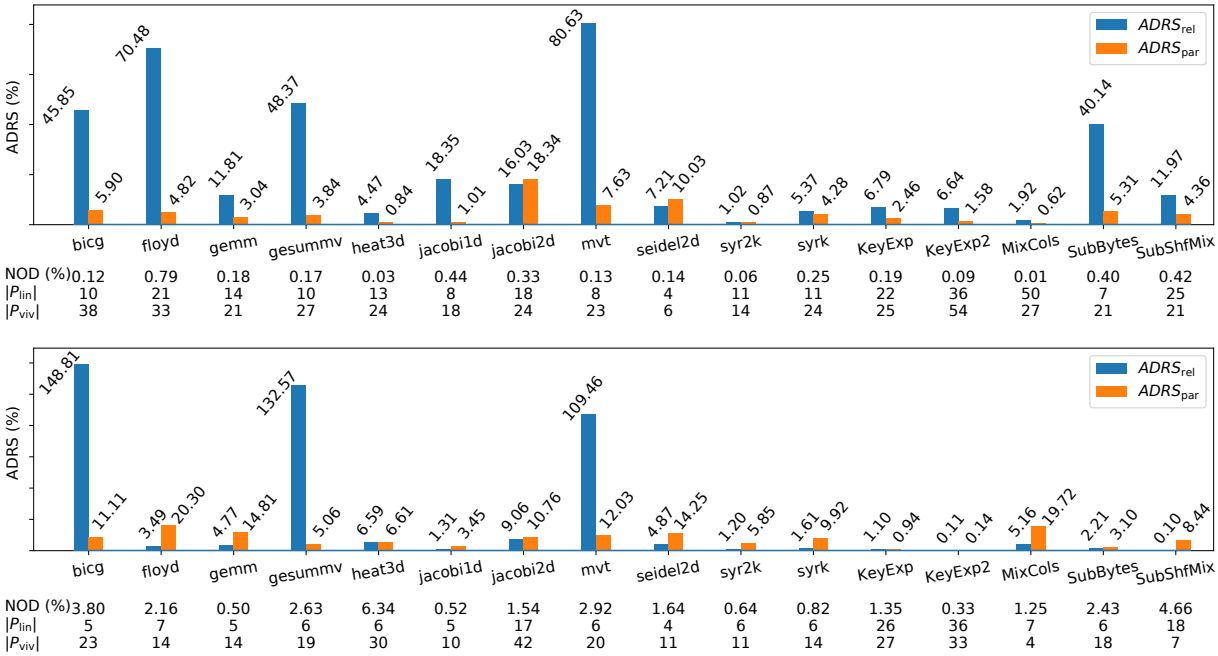


Source: Perina, Becker and Bonato (2019a).

3.4.2 Second Validation: Resource and Timing-aware Exploration

Figure 41 presents ADRS, NOD and Pareto set sizes for both `hls` and `fullsyn` experiments. Considering the `hls` experiment, the $ADRS_{\text{rel}}$ and $ADRS_{\text{par}}$ values respectively indicate that our approximations are on average 23.57% worse than the optimal and within 4.68% of the objective intervals in the Pareto sets. The average NOD value is 0.23% (all below 1%), indicating that few other design points could provide better solutions than ours. For the `fullsyn` experiment, the average $ADRS_{\text{rel}}$, $ADRS_{\text{par}}$ and NOD values are 27.03%, 9.16% and 2.1%, respectively.

Figure 41 – Values of $ADRS_{rel}$, $ADRS_{par}$, NOD, $|P_{lin}|$ and $|P_{viv}|$ for each kernel in experiments `hls` (above) and `fullsyn` (below).



Source: Perina *et al.* (2021).

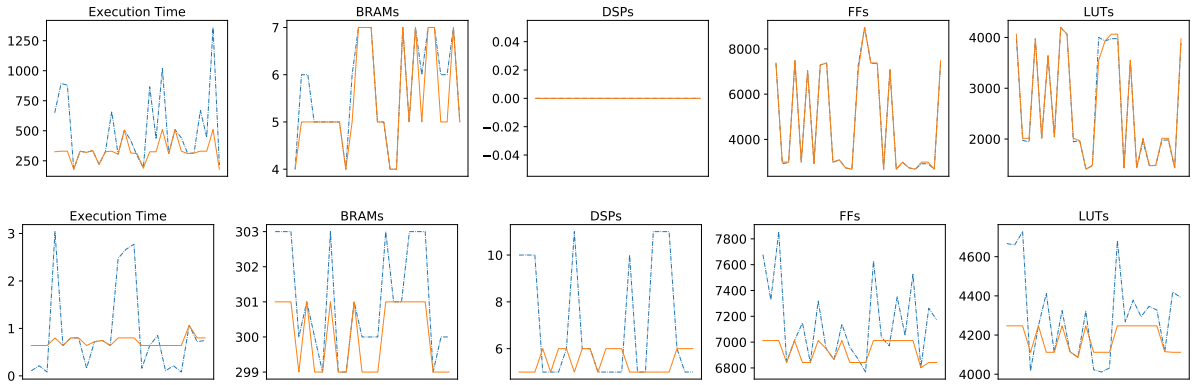
In general, the kernels in `fullsyn` present smaller $ADRS_{rel}$, larger $ADRS_{par}$ and larger NOD values when compared to `hls`. Considering that `fullsyn` is composed of smaller design spaces with less complex optimisations, the Pareto sets have shorter objective ranges. In this case, the inaccuracies of Lina are more impactful.

In most cases, our estimated Pareto sets P_{lin} are smaller than the true sets P_{viv}. We believe that this is due to: subtle resource optimisations performed by Vivado that our model does not implement (e.g. BRAM replication to alleviate port pressure); and cycle estimation issues (e.g. when partitioning is enabled, as described in (PERINA; BECKER; BONATO, 2019a)). Nonetheless, the low values of ADRS and NOD for many kernels indicate that the true Pareto points have a reasonably close approximation.

There are some kernels in both experiments with significant values of $ADRS_{rel}$, with `bicg` having the worst value. The lower plots in Figure 42 present the objective values of each point in the Pareto set for this kernel. In this case, one of the most significant contributions to $ADRS_{rel}$ comes from the execution time objective, where the leftmost point shown in the plot adds $e_{exectime} \approx 600\%$ to the calculation. It is also possible to note that our estimated Pareto points do not fully reflect the true Pareto set, however most of our solutions are still located in the lower region of the objective values. The other kernels with significant values of $ADRS_{rel}$ (e.g. `gesummv` and `mvt`) are similarly affected.

As a counter-example, the top plots in Figure 43 present the objectives for the `KeyExp2` kernel. While there are some deviations, our approximations are close, if not

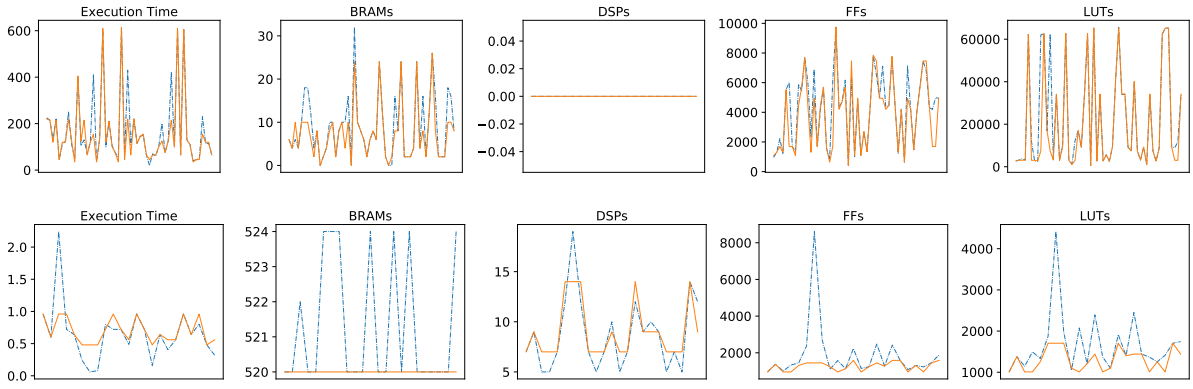
Figure 42 – Objective values for an approximation with low ADRS values (above) and high ADRS values (bottom) for the `fullsyn` experiment. The top plots represent `KeyExp`, and the bottom plots represent `bicg`. The blue dashed line represents the true Pareto points, and the continuous orange line represents the closest estimated Pareto points to each true solution. The y-axes represent the true objective value for each point (design execution time given in ns for `KeyExp` and 10^{-2}s for `bicg`).



Source: Research data.

equal, to most of the true Pareto points.

Figure 43 – Objective values for an approximation with low ADRS values (above) and high ADRS values (bottom) for the `hls` experiment. The top plots represent `KeyExp2`, and the bottom plots represent `mvt`. The blue dashed line represents the true Pareto points, and the continuous orange line represents the closest estimated Pareto points to each true solution. The y-axes represent the true objective value for each point (design execution time given in ns for `KeyExp2` and 10^{-2}s for `mvt`).



Source: Research data.

Considering all objectives, the average Mean Absolute Percentage Error (MAPE) is 49.47% for `hls` and 48.15% for `fullsyn`. Because the relative difference between the design points is more significant than the absolute values themselves for the DSE, we do not consider these error values to be an issue.

3.4.2.1 Impact of Lina Features

Considering the fastest point from the true Pareto sets, five kernels from the `hls` experiment operated at lower frequencies than the maximum explored: `gemm` (50MHz), `seidel2d` (57.14MHz), `syr2k` (57.14MHz), `syrk` (57.14MHz) and `KeyExp2` (50MHz). Lina correctly identified the first four kernels' operating frequencies, emphasising the timing constrained scheduler's importance. Without the timing awareness, the cycle count of Lina estimations would be invariant of frequency, and thus the fastest design point would always be pointed as the one with the highest frequency.

Considering the additional experiment as explained in subsubsection 3.3.3.1, the non-perfect model caused an average reduction of 3.10% and 0.87% in the $ADRS_{\text{rel}}$ and $ADRS_{\text{par}}$ values, respectively. However, the biggest benefit was noticed in the number of design points that Lina estimated as having the same shortest execution time: the number of repeated points was reduced by 66%, reflecting on a more accurate estimation of the true optimal points.

3.4.2.2 DSE Exploration Time

Table 1 shows the elapsed time to generate the trace files and estimate all design points for `hls`. We show the results without and with the use of trace cache, each with 1 or 4 concurrent executions ($p = 1$ and $p = 4$, respectively).

Table 1 – Lina DSE execution times (`hls` experiment, in s).

	Trace	w/o cache		w/ cache	
Kernel	gen.	$p = 1$	$p = 4$	$p = 1$	$p = 4$
`bicg`	11.0	186.6	102.1	136.9	66.2
`floyd`	8.2	696.1	431.6	328.2	160.9
`gemm`	16.7	2501.4	1570.0	1032.1	495.4
`gesummv`	4.2	165.0	86.1	138.1	66.5
`heat3d`	20.8	1733.3	1073.3	747.7	359.0
`jacobi1d`	0.4	22.6	10.7	22.7	10.8
`jacobi2d`	11.9	225.3	121.2	174.0	83.3
`mvt`	12.4	296.7	166.3	208.8	102.0
`seidel2d`	132.0	1615.8	986.1	633.5	291.6
`syr2k`	29.5	890.8	550.0	377.4	179.6
`syrk`	18.3	345.0	213.0	148.3	69.8
`KeyExp`	0.4	23.5	11.8	23.5	11.8
`KeyExp2`	0.4	94.0	48.1	94.1	48.1
`MixCols`	0.3	735.1	394.8	665.4	342.4
`SubBytes`	0.1	38.9	18.5	39.1	18.5
`SubShfMix`	0.2	15.7	7.5	15.7	7.5

Source: Perina *et al.* (2021).

Using $p = 4$, the exploration time is on average $\approx 2\times$ shorter than $p = 1$, which is consistent with the number of physical cores available in the i7-5500U system. With no cache and $p = 1$, `gemm` has the longest exploration time with ≈ 42 minutes. It is nonetheless faster than exploring with Vivado HLS (≈ 2 days with $p = 1$ and ≈ 12 hours with $p = 4$).

With cache enabled and $p = 4$, exploration time is reduced by an average of $3.31\times$ when compared to without cache and $p = 1$. The cache is very effective, with a hit rate of at least 98% for every kernel. The improvements are more significant in kernels with deep loop nests, where the traces can reach substantial sizes.

Considering all kernels, generating all Vivado HLS reports for the `hls` experiment took 17.81 days with $p = 1$ and 4.75 days with $p = 4$. For `fullsyn`, the total exploration took 37.39 days with $p = 1$ and 9.4 days with $p = 4$.

3.4.2.3 Comparison Analysis

We compare the speedups[11] provided by COMBA against two different results from our explorations: $F(P_{\text{lin}})$ and $F(S)$, where S is the set of all design points that Lina estimated as having the same shortest design execution time, and F is a function that returns the actual fastest point from these sets. We use Vivado HLS to generate the early cycle-accurate reports and calculate F.

Table 2 presents a comparison of the speedups achieved by COMBA and Lina. We omit the results of `rtot` and `r50` as they are similar to `r10`, apart from `jacobi1d` and `mvt` that did not synthesise on the first two experiments. For the PolyBench kernel, almost all solutions pointed by COMBA were not synthesisable. In these cases, Vivado either hung indefinitely (i.e. no response after 24 hours) or failed early due to excessive loop unroll or array partitioning. Since the resource budget of COMBA is only constrained by DSPs and BRAMs, it does not disallow the generation of circuits that overuse other resources (e.g. LUTs for multiplexers, FFs for completely partitioned arrays). COMBA estimated feasible points for all AES kernels due to their reduced loop and array sizes. For the experiments at 100MHz, the speedup was the same regardless of resource budget. For `rtot200`, the kernels are 2x faster simply due to the increased frequency.

Our approach was able to find better speedups for nearly every kernel, with the only exception being `r10`'s `jacobi1d`. It is valid to note that our design space D for this kernel did not include optimisations of the same complexity as the ones used by COMBA for this kernel. This is confirmed by the maximum achievable speedup of D: $82.81\times$.

However, the design points indicated by COMBA tend to be more resource-hungry. For `jacobi1d`, our solution uses only 5%, 24%, 30%, and 33% of the LUTs, FFs, DSPs,

[11] Considering as baseline the design point with no optimisations and with target frequency of 100MHz.

Table 2 – Speedup results from COMBA and our DSE.

	COMBA		Our DSE		
Kernel	`r10`	`rtot200`	$F(P_{\text{lin}})$	$F(S)$	$\lvert S\rvert$
`bicg`	—[b]	—[a]	2.6	5.7	64
`floyd`	—[b]	—[b]	7.0	23.5	3
`gemm`	—[a]	—[a]	5.5	5.5	81
`gesummv`	—[a]	—[a]	3.5	4.6	32
`heat3d`	—[a]	—[b]	29.0	29.0	6
`jacobi1d`	109.0	—[a]	48.5	48.5	1
`jacobi2d`	—[a]	—[b]	29.3	29.3	1
`mvt`	3.2	—[a]	3.3	6.7	144
`seidel2d`	—[a]	—[b]	1.4	1.4	10
`syr2k`	—[b]	—[b]	4.4	4.4	16
`syrk`	—[b]	—[b]	3.5	3.5	9
`KeyExp`	1.3	2.6	7.9	7.9	4
`KeyExp2`	2.8	5.6	18.9	18.9	24
`MixCols`	248.3	496.7	∞[c]	∞[c]	3
`SubBytes`	4.9	9.8	16.3	16.3	18
`SubShfMix`	4.2	8.5	34.0	6.8	2

[a] Failed due to excessive partition factor.

[b] Failed due to excessive unroll factor.

[c] Speedup of ∞ means that a combinational design was achieved.

Source: Perina *et al.* (2021).

and BRAMs used by COMBA's result, respectively. For `mvt`, we can achieve an equivalent speedup while using 5%, 3%, 67% and 21% of the same resources.

On the contrary, `KeyExp` and `KeyExp2` solutions given by $F(P_{\text{lin}})$ have a larger resource usage than COMBA's. The most significant increase is in the `KeyExp2` kernel compared to the `rtot200` variant, with a LUT increase of 28×. Such increase is expected, since our solutions are faster and we are comparing points focused on performance optimisation. In Figure 43, it is possible to note a large FF/LUT variation in the `KeyExp2`'s Pareto points.

Table 3 presents the size and exploration times of the design spaces explored. The proposed approach presents a better per-point exploration time for nearly every kernel, except for few cases in `rtot200` where COMBA's per-point exploration is faster. However, our exploration traverses several frequencies at once, whereas COMBA only allows one frequency to be explored at a time.

The speedup results in Table 2 are for the Virtex-7 platform. If we compile Lina's results on the ZCU104, there is a slight speedup increase in nearly every kernel. In `syr2k`, `syrk`, `KeyExp` and `KeyExp2`, the speedup is more noticeable: 8.47×, 6.69×, 11.07× and 35.14×, respectively. Considering all kernels (except for `MixCols` with infinite speedup),

Table 3 – Design space sizes and exploration execution times (including per-point times) for COMBA and our DSE. All times are in s.

	COMBA						Our DSE	
	# points		DSE time		Point time		Point time	
Kernel	r10	rtot200	r10	rtot200	r10	rtot200	$p=1$	$p=4$
bicg	1623	1880	918	271	0.57	0.14	0.06	0.03
floyd	158	158	60	23	0.38	0.15	0.14	0.07
gemm	804	1015	379	162	0.47	0.16	0.18	0.09
gesummv	1472	831	572	127	0.39	0.15	0.09	0.05
heat3d	334	344	729	219	2.18	0.64	0.24	0.12
jacobi1d	378	348	302	119	0.80	0.34	0.04	0.02
jacobi2d	455	459	201	89	0.44	0.19	0.12	0.06
mvt	1208	1123	753	222	0.62	0.20	0.06	0.03
seidel2d	210	246	352	111	1.68	0.45	1.59	0.88
syr2k	1037	734	632	129	0.61	0.18	0.24	0.12
syrk	805	436	348	52	0.43	0.12	0.17	0.09
KeyExp	103	103	42	11	0.41	0.11	0.07	0.04
KeyExp2	251	247	92	22	0.37	0.09	0.07	0.04
MixCols	117	126	25	8	0.21	0.06	0.07	0.03
SubBytes	164	164	4	2	0.02	0.01	0.04	0.02
SubShfMix	72	74	17	3	0.23	0.04	0.05	0.02

Source: Perina *et al.* (2021).

the average speedup is 14× and 16× for the Virtex-7 and ZCU104 platforms, respectively.

Since the Vivado HLS reports were generated for all design points in our experiments, the highest speedup achievable considering the traversed design spaces $\boldsymbol{D}$ is known. The proposed DSE was able to reach 70% of the maximum reachable speedup. It is also possible to note that the kernels with high $ADRS_{\text{rel}}$ are the ones that our exploration could not effectively reach the maximum speedup: `bicg` (24% of the maximum speedup), `floyd` (28%), `gesummv` (15%), `mvt` (12%) and `SubBytes` (16%).

3.4.3 Third Validation: Off-chip Experiments in the CNN Context

Table 4 presents the speedups achieved, where `baseline` represents the codes without optimisation directives[12], `vivbest` represents the true best design point in the considered design space[13] and `linbest` represents the best design point as estimated by Lina's DSE. Each experiment is identified by the keywords as presented in Chart 10. Although we toggle data packing between the experiments, we do not perform any manual vectorisation on the HLS code. However, we noticed that SDSoC automatically performs

[12] Apart from DDR banking, which is enabled for all design points.
[13] Acquired through running Vivado HLS for every design point.

vectorisation when DDR banking is enabled[14].

Table 4 – Performance results for each CNN exploration.

	padmemory			
	novec cons unrviv	vec cons unrviv	novec perm unrexp	vec perm unrexp
baseline†	1.0×	1.0×	1.0×	1.0×
vivbest†	241.36×	241.36×	1013.59×	1013.59×
linbest	241.36×	241.36×	563.14×	724.02×
	padlogic			
	novec cons unrviv	vec cons unrviv	novec perm unrexp	vec perm unrexp
baseline†	1.0×	1.0×	1.0×	1.0×
vivbest†	879.88×	879.88×	796.65×	796.65×
linbest	796.65×	879.88×	796.65×	710.27×

† In these cases, speedup is the same regardless of the novec/vec knob, since Vivado/SDSoC always performed automatic vectorisation. This knob only affects Lina exploration.

Source: Research data.

In the padmemory kernel, using the unrexp tool to explicitly unroll the loops brings a significant improvement: a speedup of 1013.59× was achieved considering the true best design point. Conversely, using normal HLS unroll directives lead to a maximum of 241.36×. Lina also encountered improved results when the permissive policy is used, since the explicitly unrolled codes are better reflected by this mode. Enabling data packing during Lina exploration also contributed for a better speedup. The best speedup found by Lina peaked at 724.02×, which is 71.4% of the maximum achievable speedup considering this design space (i.e. vivbest). When considering the conservative policy and no explicit unroll, both Lina explorations found the maximum achievable speedup of 241.36×.

The padlogic kernel has a slightly different scenario. The explicitly unrolled code actualy presents a worse speedup than using the SDSoC attributes. A peak of 879.88× speedup is achieved using the original HLS code. Lina is able to match this speedup when considering the conservative policy. For this kernel, the speedups found by Lina varied from 89.2% to 100.0% of the maximum achievable speedup given by vivbest.

[14] The terms "vectorisation" and "data packing" have a similar meaning in this thesis. We refer to the analysis performed by Lina as "data packing", and we refer to the optimisation performed by Vivado/SDSoC as "automatic vectorisation".

Table 5 presents the best design point configuration among the four experiments for each kernel, including their rank in the design space. Lina explorations correctly inferred the best frequency, the best loop unroll configuration and best loop pipeline configuration. It only deviated when selecting the array partition configuration. In all explorations, the best design points given by Lina was always one of the top-10 best points in the design space.

Table 5 – Configuration for the best design point for each kernel.

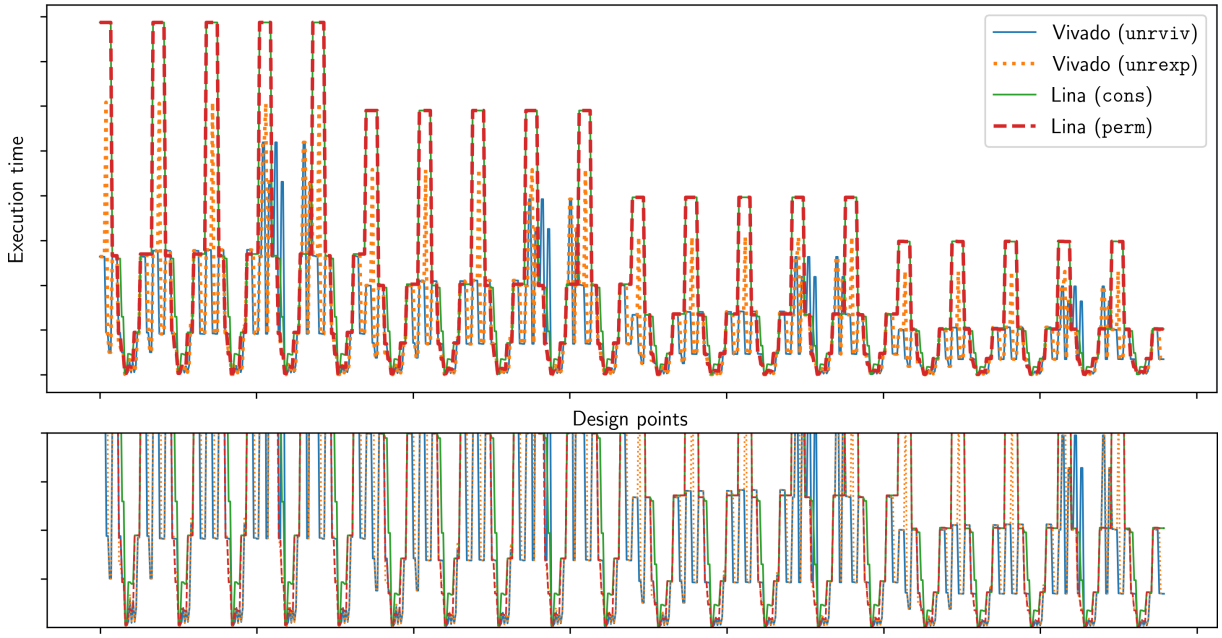
	Rank	Freq. (MHz)	Loop unroll	Loop pipeline	Partitioning of array `O`
			`padmemory`		
`baseline`	527	100.0	Off	Off	Off
`vivbest`	1	200.0	Loop level 3 factor of 2	Loop level 3	Cyclic, factor 4
`linbest`	4	200.0	Loop level 3 factor of 2	Loop level 3	Cyclic, factor 8
			`padlogic`		
`baseline`	546	100.0	Off	Off	Off
`vivbest`	1	200.0	Loop level 3 factor of 2	Loop level 3	Off
`linbest`	3	200.0	Loop level 3 factor of 2	Loop level 3	Cyclic, factor 8

Source: Research data.

Figure 44 presents a plot considering the whole design space of the `padmemory` kernel. Four design execution time subplots are shown for each design point: the values estimated by Lina using both policies and the cycle counts given by SDSoC with and without explicit unroll (`unrexp`). Banking and data packing is enabled in all cases. Although there are visible deviations between the estimated and true values, Lina characterised the lower regions of the plot — where the optimised design points are located — with good accuracy. Figure 45 presents a similar plot but for the `padlogic` kernel, which presents a similar trend as the previous one.

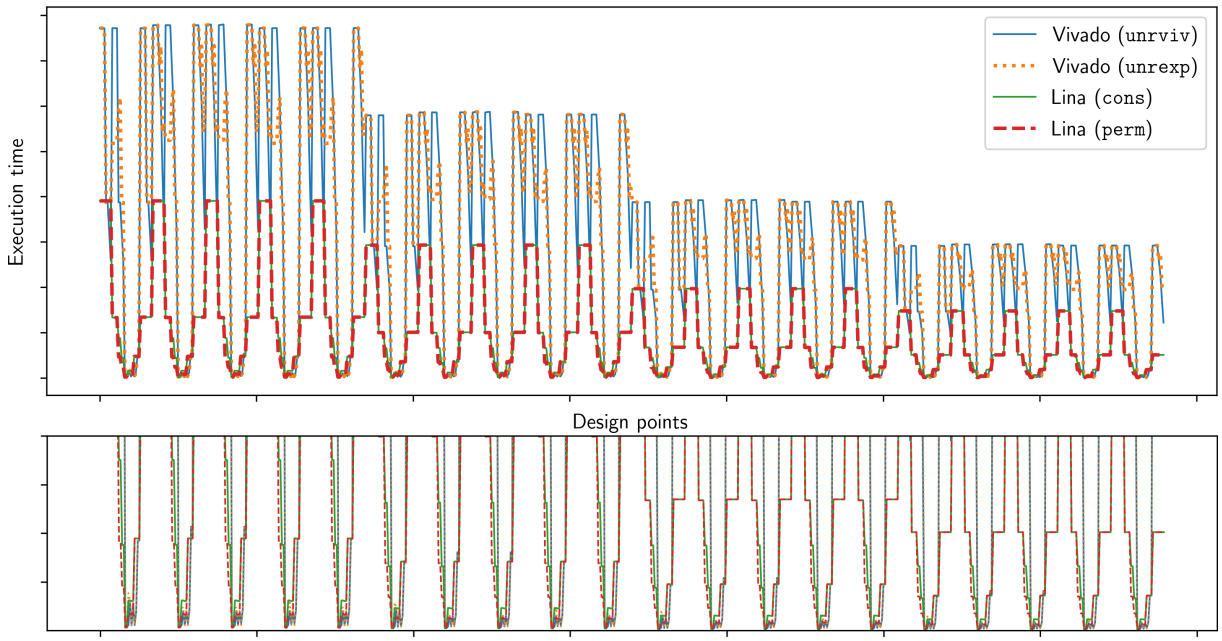
As an extra experiment, we took the best design point given by Lina for `padmemory` and we manually implemented the data vectorisation. This required a significant amount of code manipulation, since all off-chip reads and writes must be written in terms of packed data structures, while the coalescing among loop iterations must be maintained in order to allow the burst optimisations. However, only a marginal improvement was found when compared to the same design point that has no manual vectorisation: each loop iteration had its latency reduced from 162 to 157 cycles. This resulted in a 0.01% reduction of total execution time, indicating that manual vectorisation had little improvement over the automatic one performed by SDSoC for this kernel.

Figure 44 – Design execution time values and estimates for each design point in the space (`padmemory`). The plot presented at bottom is a zoomed interval of the top plot.



Source: Research data.

Figure 45 – Design execution time values and estimates for each design point in the space (`padlogic`). The plot presented at bottom is a zoomed interval of the top plot.



Source: Research data.

3.5 Final Discussion

This section presents final considerations about the presented DSE framework. First, we compare it against related work presented in Chapter 2. Then, we discuss the limitations of Lina including portability to other HLS tools. Then, a final remark is presented to close the chapter.

3.5.1 Comparison with Related Work

Our work distinguishes from the ones previously presented in the following aspects: no synthesis is required to evaluate each design point of the exploration; design frequency is supported as an exploration knob; and the resource estimation is used not only to check feasibility of each point, but also to assist on finding resource-efficient and performant designs. Chart 11 compares our DSE to those from related work.

Chart 11 – Related work comparison.

Work	General purpose	DSE requires synthesis	Frequency exploration	Optimisation objectives
Zhang *et al.* (2015)	No	Yes[a]	No	Exec. time
Choi and Cong (2018)	Yes	Yes[a]	No	Exec. time
Ferretti, Ansaloni and Pozzi (2018)	Yes	Yes	Yes	LUT, exec. time
Zhong *et al.* (2017)	Yes	No[b]	No	Exec. time
Zhao *et al.* (2019)	Yes	No	No[c]	Exec. time
This work	Yes	No	Yes	LUT, FF, DSP, BRAM and exec. time

[a] HLS results only.
[b] Synthesis required for training resource models once.
[c] COMBA supports multiple frequencies, however only one per exploration.
Source: Perina *et al.* (2021)

We chose COMBA to be the comparison tool against our approach due to: source code being publicly available; it also performs a synthesis-less DSE; it supports multiple frequencies; and it includes integer operations in the resource count. COMBA uses analytical equations to calculate the cycle count and resource footprint, and therefore it does not have a profiling overhead as our approach. Our work also differs by optimising resource footprint, whereas COMBA focuses solely on maximising performance using as many resources as needed. The clock frequency exploration of COMBA is also somewhat limited: only a single frequency is allowed per exploration execution, while Lina supports multiple frequencies per exploration. Moreover, a limited amount of frequencies are supported, whereas our DSE allows the selection of frequencies in a continuous range.

Some of the related work also provide off-chip memory analyses, however they consider different purposes and/or scenarios. Zhang *et al.* (2015) consider off-chip memory transactions in their model, however in a domain specific scenario. Zhong *et al.* (2017) provide a simple linear model for off-chip memory transactions, however they consider that all input data is transferred to on-chip memory prior to computation, and all data is retrieved afterwards. On the other hand, Lina provides a model capable of analysing the off-chip memory accesses inside the computation loop.

3.5.2 Framework Limitations

Our current implementation has some limitations. For example, there is no support to more than one loop nest per loop level, arbitrary-precision data types or variable loop bounds. Overcoming the first two limitations would require the modification of Lina internal data structures. Enabling support to variable loop bounds is also feasible, however since the DDDG generation step only uses part of the dynamic trace to perform its estimation, it could be inaccurate for kernels with irregular control flow. The DDDG generation step can be improved by sampling relevant portions of the dynamic trace in order to increase the coverage of control flow variations.

Estimating the metrics for tools other than Vivado HLS is currently not possible, although many aspects of our model are also valid for other HLS tools. Lina's scheduling outcome is sufficiently similar to the ones used by other modern HLS compilers (e.g. system of difference constraint models). For example, the initiation interval calculation used by Lina is a close approximation to the true achievable value (RAU, 1994). The variable frequency FU library is also a common aspect of many HLS compilers and can be easily disabled if a target compiler does not support it. The DSE job dispatcher is also portable since it does not use any platform-specific information to orientate the traversal. The policy of sharing FUs is also a common feature, though Lina's model is a simplistic generalisation that does not directly calculate the routing tradeoffs. At last, the array-related models and especially the equations from Makni *et al.* (2018) are quite fine-tuned for Vivado, requiring a more profound analysis on how these equations could be adapted for other tools.

The off-chip memory model is also quite specific to the Vivado HLS and SDSoC toolchains, with the greatest example being the scheduling policies. Nonetheless, most other features are common aspects of off-chip memory accesses in general. For example, it is a common practice to use coalescing and data packing to improve memory's performance. Moreover, most off-chip transactions are composed of setup and commit steps required to use the memories. We believe that most of the DDDG scheduling and optimisation parts related to the off-chip model can be reused for different compilers, whereas a more in-depth attention must be given while designing the new memory policies (or lack thereof).

3.5.3 Final Remarks

This chapter presented a design space exploration framework that estimates each design point using Lina, a fast performance and resource estimator for the Vivado HLS compiler. Large design spaces are explored in a matter of minutes, and the timing/resource awareness of the approach allows the optimisation of not only performance but also resource usage, while the clock frequency domain is also explored. The results over 16 kernels show that the estimated optimal solutions are among the 1% best solutions. An average of $14-16\times$ performance speedup is achieved, accounting for 70% of the reachable speedup considering the traversed design spaces. In comparison to another estimator (COMBA), Lina provided better speedups for nearly every kernel. Although COMBA points to more aggressive optimisations that could provide better results than ours, most of the solutions indicated were not synthesisable due to overly complex circuits.

Considering a small design space (under 100 combinations of optimisations) over 9 kernels, Lina presented smaller average relative errors when compared to its predecessor Lin-analyzer: from 16.45% to 13.01% when array partitioning is considered, and from 8.85% to 3.02% without.

We also presented the Lina's off-chip memory model that allows an exploration of memory optimisations. Aspects such as burst analysis, memory banking, data packing and HLS compiler limitations are modelled on Lina. We assessed the quality of our explorations considering off-chip accesses using a simple convolution kernel, on which speedups of at least $720\times$ were reached.

Fast and accurate synthesis-less DSE is a challenging problem, however it has the potential to provide promising efficient solutions significantly faster than approaches that rely on HLS synthesis. Such exploration can be used to optimise codes in cloud-based accelerators as a service, for example.

CHAPTER

4

CONCLUSION

This thesis presented a cycle and resource estimator for C/C++ codes targetting the Vivado HLS compiler and an accompanying design space exploration methodology. This DSE using Lina reduces the programmability complexity for a developer when using high-level synthesis, since it assists on choosing a combination of compiler directives towards better compilation outcomes (e.g. better performance).

Based on Lin-analyzer, Lina uses a dynamic trace generated from software execution to approximate the scheduling behaviour performed by the HLS compiler. Its estimation is faster than HLS compilation, and several compiler optimisation knobs are supported, allowing a fast traversal through large design spaces. For 16 C/C++ kernels explored with Lina, the estimated optimal solutions are among the 1% best options. An average of $14-16\times$ performance speedup is achieved, accounting for 70% of the reachable speedup considering the traversed design spaces. Lina also supports the exploration of off-chip memory optimisations. For a simple convolution kernel with off-chip memory accesses, Lina correctly inferred the best frequency, the best loop unroll configuration and the best loop pipeline configuration, deviating only when selecting the array partition configuration. The best points given by Lina were always one of the top-10 best points in the design space, reaching speedups of at least $720\times$.

The first approaches on this thesis targetted a model capable of estimating the most suitable platform between FPGA and GPU for a given OpenCL kernel. Two machine learning models were implemented to this end. The first approach used a data mining clusterisation tool and was able to reach an average hit rate of 70% to 88% when estimating the most suitable platform for a kernel. The second approach was modelled around neural networks, and was able to reach nearly 85% of hit rate. We noticed, however, that every kernel used for training and validation were extremely suited for the platform they were initially designed for. Considering that most OpenCL kernels available are designed using the SIMD execution model targetting GPUs, we then gathered

several OpenCL kernels using this model, and we evaluated their performance on FPGA and GPU considering different optimisation efforts. Results have shown that a significant amount of optimisation is necessary to bring FPGA closer to a competitive level for existing OpenCL kernels, often leading to complete code rewrite. This led to our decision on shifting to C/C++ and implementing Lina instead.

At last, we present another FPGA-GPU comparative analysis. In the FPGA side, we optimise the applications using Lina. On the GPU side, we use CUDA variants for the same applications. Results have shown that our approach loses in both performance and energy consumption when compared against GPUs. We also noticed that in the FPGA side, the speedup of the computation kernels isolated were greater than the total speedup achieved when considering the whole application execution. Since Lina only optimises the computation loop, there is still a significant overhead on the data transfers from/to the FPGA device. As a final analysis, we present an extrapolation considering a hypothetical situation where these overheads were to be mitigated, i.e. the total speedup achieved is the same as the speedup achieved by the computation part alone. In this case, the solutions given by Lina would be better than the GPU counterparts for two kernels. While the FPGA is on the disadvantage in overall, it must be noted that we are comparing applications that were automatically optimised by Lina against codes that were manually tailored for GPUs using CUDA.

Although our approach is quite fine-tuned in several aspects to Vivado HLS and SDSoC, our methodology has further shown the benefits of using dynamic traces for decision making when performing DSE. For example, the memory accesses are already resolved by the trace, not requiring any static complex inferring that could incur in larger estimation times. Our results have shown that such approach is able to deliver optimised solutions considering several scenarios.

We believe that our approach using Lina fulfills the main objective, of **providing a fast optimisation approach that automatically improves the quality of FPGA designs generated from HLS tools.** The main contributions derived from this thesis are:

- The **timing-aware model** of Lina, that allows several frequencies to be explored and a more precise in-cycle scheduling;
- The **non-perfect loop analyser**, that improves Lina's accuracy when estimating loop nests that are not perfect;
- The **resource-aware model**, that allows Lina to estimate the resource usage of the design points. This model can be used to find points that are optimised not only in design execution time, but also in resource usage;

- The **off-chip memory model**, which allows the estimation of kernels accessing off-chip memory. This model includes optimisations such as burst analysis, memory banking, data packing and scheduling policies.

Other contributions include several tools that were developed during this thesis, such as the `vivado-fsmgem`, ProfCounter, `zynprof`, among others as presented in Appendix A. These are not directly related to the main contributions of Lina, however they were of great support and could be further useful in future circumstances.

There are several potential improvements for this thesis. Possible future work include:

- Addition of more features in Lina, such as support to variable loops, more complex data structures, etc.;
- Several aspects of the Lina code require refactor and optimisation, which could further reduce its estimation and exploration time;
- Migration to newer versions of Vivado/Vitis HLS;
- Lina currently considers that each array is completely located on or off-chip. Support to buffering optimisations such as tiling, caching, etc. could bring further exploration possibilities;
- Our last FPGA-GPU comparison results have shown that there is still a significant overhead on data transfers from/to the FPGA device. Future experiments could include the optimisation of the whole application, for example by interleaving data transfers and kernel execution;
- At last, the whole DSE framework here presented is in a prototypal stage. Several modifications are still required on the input applications in order to make them compatible, for example by adapting Makefiles or creating the OpenCL kernel templates. Most of these modifications could be automatically performed, for example by an LLVM transformation pass. Thus, a more seamless experience for the exploration approach is desirable.

In conclusion, while our model assists on reducing the hardware burden when programming FPGAs via HLS, it is still far from the software-based experience when working with high-level platforms. We encountered several issues with the HLS compiler that hindered our results. For example, the `unrexp` tool that we created to explicitly unroll loops when the automatic HLS unroll disabled off-chip coalescing optimisations. While this tool allowed the HLS compiler to correctly infer some of the missed optimisations, it also significantly increased the HLS compilation complexity due to the lengthy unrolled source

codes, which often lead to compilation failure. All these aspects altogether expose the gap that still exists in the high-level compilation field for FPGAs. For efficient optimised FPGA designs, programming with RTL languages is still the best approach despite the extensive hardware expertise and man-hours required. Nonetheless, we acknowledge the importance of High-Level Synthesis compilers as they broaden the accessibility of FPGAs to communities that would never dive in RTL programming. As presented in this thesis and other related work, the research community is actively in search of a better high-level environment for FPGAs.

BIBLIOGRAPHY

AGARWAL, A.; NG, M. C. *et al.* A Comparative Evaluation of High-Level Hardware Synthesis Using Reed–Solomon Decoder. **IEEE Embedded Systems Letters**, IEEE, v. 2, n. 3, p. 72–76, 2010. Citations on pages 142 and 158.

AUSTIN, T. M.; SOHI, G. S. Dynamic Dependency Analysis of Ordinary Programs. In: **Proceedings of the 19th annual international symposium on Computer architecture**. [S.l.: s.n.], 1992. p. 342–351. Citation on page 51.

BILAVARN, S.; GOGNIAT, G.; PHILIPPE, J.-L.; BOSSUET, L. Design Space Pruning Through Early Estimations of Area/Delay Tradeoffs for FPGA Implementations. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 25, n. 10, p. 1950–1968, 2006. Citations on pages 43 and 51.

BJUREUS, P.; MILLBERG, M.; JANTSCH, A. FPGA Resource and Timing Estimation from Matlab Execution Traces. In: IEEE. **Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No. 02TH8627)**. [S.l.], 2002. p. 31–36. Citations on pages 43 and 51.

BORKAR, S. The Exascale Challenge. In: IEEE. **Proceedings of 2010 International Symposium on VLSI Design, Automation and Test**. [S.l.], 2010. p. 2–3. Citations on pages 13 and 35.

CANIS, A.; CHOI, J.; ALDHAM, M.; ZHANG, V.; KAMMOONA, A.; ANDERSON, J. H.; BROWN, S.; CZAJKOWSKI, T. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. In: **Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays**. [S.l.: s.n.], 2011. p. 33–36. Citation on page 41.

CHE, S.; BOYER, M.; MENG, J.; TARJAN, D.; SHEAFFER, J. W.; LEE, S.-H.; SKADRON, K. Rodinia: A Benchmark Suite for Heterogeneous Computing. In: IEEE. **Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on**. [S.l.], 2009. p. 44–54. Citation on page 42.

CHOI, J. **LegUp 5.1 is released!** 2017. Available at <https://www.legupcomputing.com/blog/index.php/2017/07/06/legup-5-1-is-released/>, accessed 7th feb. 2022. Citation on page 42.

CHOI, Y.-k.; CONG, J. HLS-Based Optimization and Design Space Exploration for Applications with Variable Loop Bounds. In: IEEE. **2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [S.l.], 2018. p. 1–8. Citations on pages 44 and 113.

CZYŻAK, P.; JASZKIEWICZ, A. Pareto Simulated Annealing — A Metaheuristic Technique for Multiple-Objective Combinatorial Optimization. **Journal of Multi-Criteria Decision Analysis**, Wiley Online Library, v. 7, n. 1, p. 34–47, 1998. Citation on page 88.

DANALIS, A.; MARIN, G.; MCCURDY, C.; MEREDITH, J. S.; ROTH, P. C.; SPAFFORD, K.; TIPPARAJU, V.; VETTER, J. S. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In: ACM. **Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units**. [S.l.], 2010. p. 63–74. Citation on page 42.

DENNARD, R. H.; GAENSSLEN, F. H.; YU, H.-N.; RIDEOUT, V. L.; BASSOUS, E.; LEBLANC, A. R. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. **IEEE Journal of solid-state circuits**, IEEE, v. 9, n. 5, p. 256–268, 1974. Citation on page 34.

ENZLER, R.; JEGER, T.; COTTET, D.; TRÖSTER, G. High-Level Area and Performance Estimation of Hardware Building Blocks on FPGAs. In: SPRINGER. **International Workshop on Field Programmable Logic and Applications**. [S.l.], 2000. p. 525–534. Citations on pages 43 and 51.

FERRETTI, L.; ANSALONI, G.; POZZI, L. Lattice-Traversing Design Space Exploration for High Level Synthesis. In: IEEE. **2018 IEEE 36th International Conference on Computer Design (ICCD)**. [S.l.], 2018. p. 210–217. Citations on pages 44 and 113.

GAO, X.; WICKERSON, J.; CONSTANTINIDES, G. A. Automatically Optimizing the Latency, Area, and Accuracy of C Programs for High-Level Synthesis. In: **Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays**. [S.l.: s.n.], 2016. p. 234–243. Citation on page 67.

Geeks3D. **Graphics Cards Thermal Design Power (TDP) Database**. 2015. Available at <https://www.geeks3d.com/20090618/graphics-cards-thermal-design-power-tdp-database/>, accessed 7th feb. 2022. Citation on page 162.

GIEFERS, H.; STAAR, P.; BEKAS, C.; HAGLEITNER, C. Analyzing the Energy-Efficiency of Sparse Matrix Multiplication on Heterogeneous Systems: A Comparative Study of GPU, Xeon Phi and FPGA. In: IEEE. **2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**. [S.l.], 2016. p. 46–56. Citation on page 42.

HUTTON, M. Understanding How the New Intel(r) HyperFlex(tm) Architecture Enables Next-Generation High-Performance Systems. **Altera White Paper**, 2015. Citation on page 36.

Intel Corporation. **Arria 10 Power Reference Design**. 2018. Available at <https://www.intel.com/content/www/us/en/programmable/products/reference-designs/all-reference-designs/power/arria-10-power-ref-design.html>, accessed 7th feb. 2022. Citation on page 162.

_____. **Hardware Accelerator Research Program**. 2018. Available at <https://software.intel.com/en-us/hardware-accelerator-research-program>, accessed 6th nov. 2018. Citation on page 163.

JIANG, J.; WANG, Z.; LIU, X.; GÓMEZ-LUNA, J.; GUAN, N.; DENG, Q.; ZHANG, W.; MUTLU, O. Boyi: A Systematic Framework for Automatically Deciding the Right Execution Model of OpenCL Applications on FPGAs. In: **Proceedings of the 2020**

ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. [S.l.: s.n.], 2020. p. 299–309. Citation on page 45.

KANDURI, A.; RAHMANI, A. M.; LILJEBERG, P.; HEMANI, A.; JANTSCH, A.; TENHUNEN, H. A Perspective on Dark Silicon. In: **The Dark Side of Silicon**. [S.l.]: Springer, 2017. p. 3–20. Citations on pages 34 and 35.

KONO, F.; NAKASATO, N.; HAYASHI, K.; VAZHENIN, A.; SEDUKHIN, S. Evaluations of OpenCL-written tsunami simulation on FPGA and comparison with GPU implementation. **The Journal of Supercomputing**, Springer, v. 74, n. 6, p. 2747–2775, 2018. Citation on page 42.

KULKARNI, D.; NAJJAR, W. A.; RINKER, R.; KURDAHI, F. J. Compile-Time Area Estimation for LUT-Based FPGAs. **ACM Transactions on Design Automation of Electronic Systems (TODAES)**, ACM New York, NY, USA, v. 11, n. 1, p. 104–122, 2006. Citations on pages 43 and 51.

LARSEN, E. S.; MCALLISTER, D. Fast Matrix Multiplies using Graphics Hardware. In: **Proceedings of the 2001 ACM/IEEE Conference on Supercomputing**. [S.l.: s.n.], 2001. p. 55–55. Citation on page 34.

LI, P.; ZHANG, P.; POUCHET, L.-N.; CONG, J. Resource-Aware Throughput Optimization for High-Level Synthesis. In: **Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays**. [S.l.: s.n.], 2015. p. 200–209. Citation on page 54.

LIANG, Y.; WANG, S.; ZHANG, W. FlexCL: A Model of Performance and Power for OpenCL Workloads on FPGAs. **IEEE Transactions on Computers**, IEEE, v. 67, n. 12, p. 1750–1764, 2018. Citation on page 45.

MAKNI, M.; NIAR, S.; BAKLOUTI, M.; ABID, M. HAPE: A high-level area-power estimation framework for FPGA-based accelerators. **Microprocessors and Microsystems**, Elsevier, v. 63, p. 11–27, 2018. Citations on pages 68, 69, 70, 71, and 114.

MARTINS, L. G. A. **Exploration of optimization sequences of the compiler based on hybrid techniques of complex data mining**. Phd Thesis (PhD Thesis) — University of São Paulo, 2015. Citation on page 140.

Micron Technology, Inc. **GDDR5 SGRAM Introduction**. 2014. Available at <https://www.micron.com/-/media/client/global/documents/products/technical-note/dram/tned01_gddr5_sgram_introduction.pdf>, accessed 7th feb. 2022. Citation on page 75.

MOORE, G. E. **Cramming more components onto integrated circuits**. [S.l.]: McGraw-Hill New York, 1965. Citation on page 34.

MUSLIM, F. B.; MA, L.; ROOZMEH, M.; LAVAGNO, L. Efficient FPGA Implementation of OpenCL High-Performance Computing Applications via High-Level Synthesis. **IEEE Access**, IEEE, v. 5, p. 2747–2762, 2017. Citations on pages 38, 42, 157, 158, 160, and 164.

Nallatech. **Nallatech 385 – with Stratix V A7 FPGA**. 2018. Available at <https://www.nallatech.com/store/fpga-accelerated-computing/pcie-accelerator-cards/385-a7/>, accessed 6th nov. 2018. Citation on page 162.

_____. **Nallatech 385A FPGA Accelerator Card**. 2018. Available at <https://www.nallatech.com/store/fpga-accelerated-computing/pcie-accelerator-cards/nallatech-385a-arria10-1150-fpga/>, accessed 6th nov. 2018. Citation on page 162.

NANE, R.; SIMA, V.-M.; PILATO, C.; CHOI, J.; FORT, B.; CANIS, A.; CHEN, Y. T.; HSIAO, H.; BROWN, S.; FERRANDI, F. *et al.* A Survey and Evaluation of FPGA High-Level Synthesis Tools. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 35, n. 10, p. 1591–1604, 2015. Citation on page 41.

NDU, G.; NAVARIDAS, J.; LUJÁN, M. CHO: Towards a Benchmark Suite for OpenCL FPGA Accelerators. In: **Proceedings of the 3rd International Workshop on OpenCL**. [S.l.: s.n.], 2015. p. 1–10. Citation on page 142.

NVIDIA Corporation. **NVIDIA Quadro K620**. 2018. Available at <https://nvidiastore.com.br/nvidia-quadro-k620>, accessed 7th feb. 2022. Citation on page 162.

OPPERMANN, J.; SOMMER, L.; WEBER, L.; REUTER-OPPERMANN, M.; KOCH, A.; SINNEN, O. SkyCastle: A Resource-Aware Multi-Loop Scheduler for High-Level Synthesis. In: IEEE. **2019 International Conference on Field-Programmable Technology (ICFPT)**. [S.l.], 2019. p. 36–44. Citation on page 45.

PALERMO, G.; SILVANO, C.; ZACCARIA, V. ReSPIR: A Response Surface-Based Pareto Iterative Refinement for Application-Specific Design Space Exploration. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 28, n. 12, p. 1816–1829, 2009. Citation on page 88.

PEEMEN, M.; SETIO, A. A.; MESMAN, B.; CORPORAAL, H. Memory-Centric Accelerator Design for Convolutional Neural Networks. In: IEEE. **2013 IEEE 31st International Conference on Computer Design (ICCD)**. [S.l.], 2013. p. 13–19. Citation on page 96.

PERINA, A. B.; BECKER, J.; BONATO, V. Lina: Timing-Constrained High-Level Synthesis Performance Estimator for Fast DSE. In: IEEE. **2019 International Conference on Field-Programmable Technology (ICFPT)**. [S.l.], 2019. p. 343–346. Citations on pages 47, 101, 102, 103, 104, and 131.

_____. Profcounter: Line-Level Cycle Counter for Xilinx OpenCL High-Level Synthesis. In: IEEE. **2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)**. [S.l.], 2019. p. 618–621. Citation on page 130.

PERINA, A. B.; BONATO, V. Mapping Estimator for OpenCL Heterogeneous Accelerators. In: IEEE. **2018 International Conference on Field-Programmable Technology (FPT)**. [S.l.], 2018. p. 294–297. Citations on pages 130, 149, 150, 151, and 153.

PERINA, A. B.; SILITONGA, A.; BECKER, J.; BONATO, V. Fast Resource and Timing Aware Design Optimisation for High-Level Synthesis. **IEEE Transactions on Computers**, IEEE, v. 70, n. 12, p. 2070–2082, 2021. Citations on pages 38, 48, 52, 55, 56, 60, 61, 64, 90, 94, 95, 104, 106, 108, 109, 113, 132, and 174.

POUCHET, L.-N. **PolyBench: the Polyhedral Benchmark Suite**. 2012. Available at <http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/>, accessed 7th feb. 2022. Citation on page 42.

PU, Y.; PENG, J.; HUANG, L.; CHEN, J. An efficient KNN algorithm implemented on FPGA based heterogeneous computing system using OpenCL. In: IEEE. **2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines**. [S.l.], 2015. p. 167–170. Citation on page 42.

RAMANATHAN, N.; CONSTANTINIDES, G. A.; WICKERSON, J. Concurrency-Aware Thread Scheduling for High-Level Synthesis. In: IEEE. **2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)**. [S.l.], 2018. p. 101–108. Citation on page 42.

RAU, B. R. Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops. In: **Proceedings of the 27th annual international symposium on Microarchitecture**. [S.l.: s.n.], 1994. p. 63–74. Citations on pages 54 and 114.

ROSA, L. S.; BOUGANIS, C.-S.; BONATO, V. Scaling Up Modulo Scheduling for High-Level Synthesis. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 38, n. 5, p. 912–925, 2018. Citation on page 42.

ROSER, M.; RITCHIE, H. Technological Progress. **Our World in Data**, 2013. Https://ourworldindata.org/technological-progress. Citation on page 33.

SANCHES, A.; CARDOSO, J. M.; DELBEM, A. C. Identifying Merge-Beneficial Software Kernels for Hardware Implementation. In: IEEE. **2011 International Conference on Reconfigurable Computing and FPGAs**. [S.l.], 2011. p. 74–79. Citation on page 138.

SEIFOORI, Z.; EBRAHIMI, Z.; KHALEGHI, B.; ASADI, H. Introduction to Emerging SRAM-Based FPGA Architectures in Dark Silicon Era. In: **Advances in Computers**. [S.l.]: Elsevier, 2018. v. 110, p. 259–294. Citation on page 36.

Semiconductor Industry Association. **International Technology Roadmap for Semiconductors - 2013 Edition**. 2013. Available at <https://www.semiconductors.org/resources/2013-international-technology-roadmap-for-semiconductors-itrs/>, accessed 7th feb. 2022. Citations on pages 13 and 35.

SHAO, Y. S.; REAGEN, B.; WEI, G.-Y.; BROOKS, D. Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures. In: IEEE. **2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)**. [S.l.], 2014. p. 97–108. Citations on pages 43 and 51.

SILITONGA, A.; SCHADE, F.; JIANG, G.; BECKER, J. HLS-based Performance and Resource Optimization of Cryptographic Modules. In: IEEE. **2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)**. [S.l.], 2018. p. 1009–1016. Citation on page 93.

STOUTCHININ, A.; CONTI, F.; BENINI, L. Optimally Scheduling CNN Convolutions for Efficient Memory Access. **arXiv preprint arXiv:1902.01492**, 2019. Citation on page 96.

STRATTON, J. A.; RODRIGUES, C.; SUNG, I.-J.; OBEID, N.; CHANG, L.-W.; ANSSARI, N.; LIU, G. D.; HWU, W.-m. W. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. **Center for Reliable and High-Performance Computing**, v. 127, p. 29, 2012. Citation on page 42.

SUMOYAMA, A. S. **Classifier of kernels for hybrid computing platform mapping composed by FPGA and GPP**. Master's Thesis (Master's Thesis) — University of São Paulo, 2016. Citations on pages 138 and 141.

TATSUMI, S.; HARIYAMA, M.; MIURA, M.; ITO, K.; AOKI, T. OpenCL-Based Design of an FPGA Accelerator for Phase-Based Correspondence Matching. In: THE STEERING COMMITTEE OF THE WORLD CONGRESS IN COMPUTER SCIENCE, COMPUTER …. **Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)**. [S.l.], 2015. p. 90. Citation on page 42.

WELLER, D.; OBORIL, F.; LUKARSKI, D.; BECKER, J.; TAHOORI, M. Energy Efficient Scientific Computing on FPGAs using OpenCL. In: ACM. **Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays**. [S.l.], 2017. p. 247–256. Citations on pages 38, 42, 157, 158, 160, and 164.

Wikipedia. **Transistor Count**. 2022. Available at <https://en.wikipedia.org/wiki/Transistor_count>, accessed 7th feb. 2022. Citation on page 33.

WILLIAMS, S.; WATERMAN, A.; PATTERSON, D. Roofline: An Insightful Visual Performance Model for Multicore Architectures. **Communications of the ACM**, ACM, v. 52, n. 4, p. 65–76, 2009. Citation on page 149.

Xilinx, Inc. **Xilinx Acquires AutoESL to Enable Designer Productivity and Innovation With FPGAs and Extensible Processing Platform**. 2011. Available at <https://www.prnewswire.com/news-releases/xilinx-acquires-autoesl-to-enable-designer-productivity-and-innovation-with-fpgas-and-\extensible-processing-platform-114922409.html>, accessed 7th feb. 2022. Citation on page 41.

_____. **ZCU104 Evaluation Board - User Guide**. 2018. Available at <https://www.xilinx.com/support/documentation/boards_and_kits/zcu104/ug1267-zcu104-eval-bd.pdf>, accessed 7th feb. 2022. Citation on page 176.

ZHANG, C.; LI, P.; SUN, G.; GUAN, Y.; XIAO, B.; CONG, J. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In: **Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays**. [S.l.: s.n.], 2015. p. 161–170. Citations on pages 44, 96, 113, and 114.

ZHAO, J.; FENG, L.; SINHA, S.; ZHANG, W.; LIANG, Y.; HE, B. Performance Modeling and Directives Optimization for High Level Synthesis on FPGA. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, 2019. Citations on pages 45, 67, and 113.

ZHONG, G.; PRAKASH, A.; LIANG, Y.; MITRA, T.; NIAR, S. Lin-analyzer: a High-level Performance Analysis Tool for FPGA-based Accelerators. In: IEEE. **2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)**. [S.l.], 2016. p. 1–6. Citations on pages 37, 38, 43, 47, 51, 54, 68, and 71.

ZHONG, G.; PRAKASH, A.; WANG, S.; LIANG, Y.; MITRA, T.; NIAR, S. Design Space Exploration of FPGA-based Accelerators with Multi-level Parallelism. In: IEEE. **Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017**. [S.l.], 2017. p. 1141–1146. Citations on pages 44, 113, and 114.

ZHONG, G.; VENKATARAMANI, V.; LIANG, Y.; MITRA, T.; NIAR, S. Design Space Exploration of Multiple Loops on FPGAs using High Level Synthesis. In: IEEE. **2014 IEEE 32nd international conference on computer design (ICCD)**. [S.l.], 2014. p. 456–463. Citation on page 88.

ZOHOURI, H. R. **High Performance Computing with FPGAs and OpenCL**. Phd Thesis (PhD Thesis) — Tokyo Institute of Technology, 2018. Citations on pages 42, 157, 158, 160, 161, 163, 164, and 170.

ZOHOURI, H. R.; MARUYAMA, N.; SMITH, A.; MATSUDA, M.; MATSUOKA, S. Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs. In: IEEE PRESS. **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. [S.l.], 2016. p. 35. Citations on pages 38 and 158.

APPENDIX

A

PUBLISHED MATERIAL AND DEVELOPED TOOLS

A.1 Published Material

The following publications were derived from this thesis:

- **Journal paper:**
  - PERINA, André B. et al. Fast Resource and Timing Aware Design Optimisation for High-Level Synthesis. **IEEE Transactions on Computers**, v. 70, n. 12, p. 2070-2082, 2021.
- **In conference proceedings:**
  - PERINA, André Bannwart; BONATO, Vanderlei. Mapping Estimator for OpenCL Heterogeneous Accelerators. In: **2018 International Conference on Field-Programmable Technology (FPT)**. IEEE, 2018. p. 294-297.
  - PERINA, André Bannwart; BECKER, Jürgen; BONATO, Vanderlei. Profcounter: Line-level cycle counter for xilinx opencl high-level synthesis. In: **2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)**. IEEE, 2019. p. 618-621.
  - PERINA, André Bannwart; BECKER, Jürgen; BONATO, Vanderlei. Lina: Timing-constrained high-level synthesis performance estimator for fast DSE. In: **2019 International Conference on Field-Programmable Technology (ICFPT)**. IEEE, 2019. p. 343-346.

A.2 Developed Tools

The following tools and repositories were developed as part of this research (presented in chronological order):

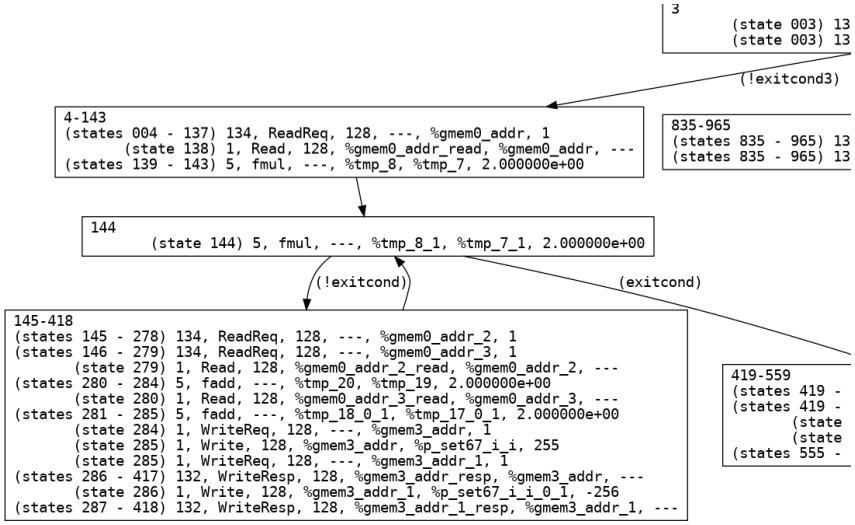
- **`hostcodegen`: OpenCL Host Code Generator**
  - This script generates template OpenCL host codes for kernel testbenching;
  - The kernel is described using an XML file containing details about inputs, outputs, preprocessing, postprocessing, etc.;
  - The host codes from Appendix C were generated using this tool;
  - Repository: <https://github.com/comododragon/hostcodegen>;
- **`mdamicore2`: DAMICORE Expanded Version for Multiple Representations Version 2**
  - This tool invokes DAMICORE for several representations and manages them accordingly;
  - Main data mining tool used in section B.1;
  - Uses a Python-based implementation of DAMICORE (<https://gitlab.com/comododragon/damicorepy>). This is a fork of an outdated Python 2 version (<https://gitlab.com/adaptsys/damicorepy>).
- **OpCount LLVM Pass** (PERINA; BONATO, 2018)
  - This is an LLVM pass that can be used to count several static metrics from an OpenCL kernel;
  - OpCount was used to generate the dataset presented in section B.2;
  - Repository: <https://github.com/comododragon/opcount>;
- **`opencl-4all`**
  - Repository containing all explored kernels and all analysis spreadsheets related to Appendix C;
  - Repository: <https://github.com/comododragon/opencl-4all>;
- **SDx OpenCL ProfCounter** (PERINA; BECKER; BONATO, 2019b)
  - Active cycle counter for OpenCL kernels using Xilinx SDx (either SDSoC or SDAccel);
  - Composed of an additional OpenCL kernel that can be added to a project;

  - This kernel contains a live cycle counter, and records timestamps based on commands sent by the main kernel;
  - ProfCounter was used as an analysis tool for early FU characterisation;
  - Repository: <https://github.com/comododragon/sdx-ocl-profcounter>;

- **Lina (first version)** (PERINA; BECKER; BONATO, 2019a)
  - This is the version of Lina used in the first validation experiment (subsection 3.3.2);
  - This version does not include the cache mechanism, resource awareness, or the off-chip memory model;
  - Repository: <https://github.com/comododragon/lina/tree/d85c4a49019027a41970b5e11aa14558951efe35>[1];

- **`vivado-fsmgen`: Vivado/Vitis HLS FSM Diagram Generator**
  - This is a script that parses a Vivado HLS report and generates the respective finite state machine diagram for it. Figure 46 shows part of an example FSM generated by this tool;

Figure 46 – Part of an FSM generated by `vivado-fsmgen`.



Source: Elaborated by the author.

  - LLVM IR instructions can be filtered out, so that only the ones of interest are shown;
  - This tool has been extensively used to study the timing analysis of Vivado. It has also been used to study how the off-chip transactions were allocated;

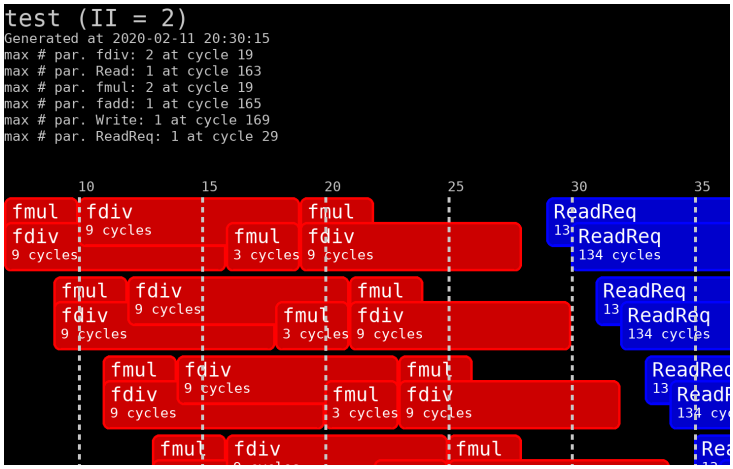
[1] This version is now deprecated.

  - Repository: <https://github.com/comododragon/vivado-fsmgen>;

- **`pipelook`: Pipeline Diagram Generator for Vivado/Vitis HLS Designs**

  - This is a script that generates the pipeline schedule from a Vivado HLS report. Figure 47 shows part of a diagram generated by this tool;

Figure 47 – Part of a scheduled pipeline as presented by `pipelook`.



Source: Elaborated by the author.

  - The II value can be adjusted in order to evaluate invalid or bottlenecking configurations;
  - This tool was extensively used to understand how II was calculated when multiple off-chip transactions are considered (see paragraph 3.2.4.3.2);
  - Repository: bundled with `vivado-fsmgen`;

- **Lina (resource-aware version)** (PERINA *et al.*, 2021)

  - This is an updated version of Lina that includes resource awareness, cache mechanism and other quality improvements;
  - Does not include the off-chip memory model;
  - Repository: <https://github.com/comododragon/lina>;

- **Lina (resource-aware + off-chip version)**

  - This version has the off-chip memory model attached;
  - Repository: <https://github.com/comododragon/linaii>;

- **`nvpm`: NVIDIA Power Monitor**

  - This small C application uses the NVIDIA's Management Library (NVML) to acquire power sensors information periodically;

  - Then, `nvpm` synchronises the sensed values with the period that the kernel was executed;
  - The consumed energy during kernel execution is then calculated;
  - Used to sense the NVIDIA GPU on Appendix E;

- **`zynprof`: Zynq Profiler**
  - Small suite used to calculate the consumed energy when running kernels on a Xilinx Zynq UltraScale+ board;
  - It is composed of a Python script and a C program. The C program is executed on an Arduino Nano microcontroller, which communicates with the power sensors;
  - The Python script peridocally polls the microcontroller and acquires the power information;
  - Similarly to `nvpm`, `zynprof` synchronises the sensed values with the period that the kernel was executed;
  - The consumed energy during kernel execution is then calculated;
  - Used to sense the ZCU104 FPGA on Appendix E.

APPENDIX

B

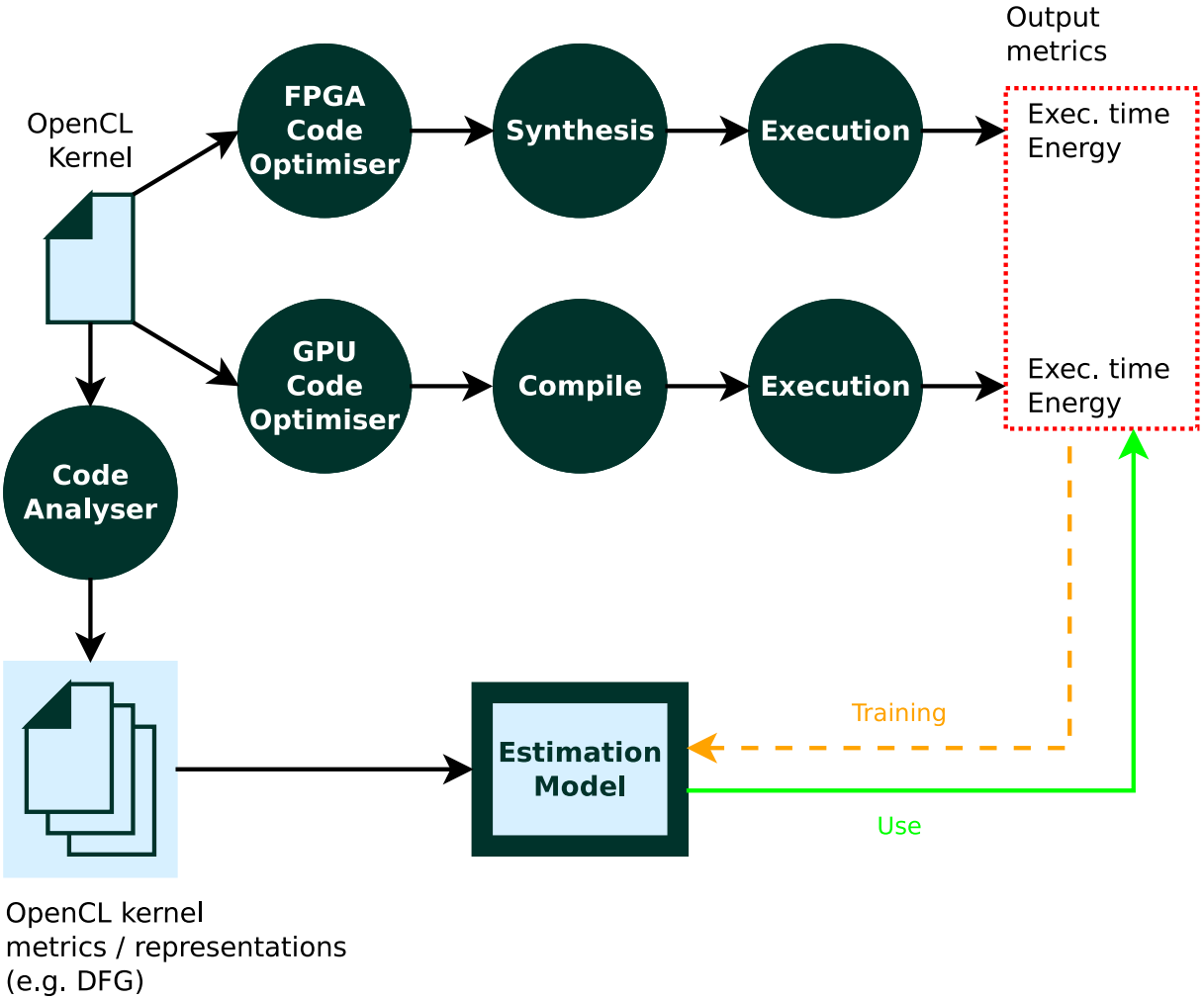
EARLY APPROACHES USING MACHINE LEARNING MODELS

This appendix presents our early attempts of a mapping estimator for OpenCL kernels using machine learning. The intent was to provide a fast estimation on where a given OpenCL kernel would run best: FPGA or GPU. This rapid insight could be useful in heterogeneous cloud systems for a fast and efficient deployment of OpenCL services.

We also proposed the inclusion of a kernel optimiser that would explore and optimise the kernels for FPGA (and if possible for GPUs). Our early machine learning model would then take this optimiser in consideration when estimating the best platform for an input kernel. Figure 48 presents the concept of our early proposed model. From now on, we will refer to an input kernel that is being tested by our model as a Kernel Under Test (KUT).

Two approaches were developed. The first one uses a data mining tool named DAMICORE and is presented in section B.1. The second approach uses neural networks and is presented in section B.2.

Figure 48 – Abstract representation of our proposed model. First, a reference set of OpenCL kernels is optimised, executed and profiled. A code analyser is then used to derive other representations (e.g. DFG) and/or numerical metrics from each kernel. These, along with the profiled output metrics, are used to train the estimation model (dashed orange arrow). During the use phase (solid green arrow), a test OpenCL kernel (KUT) is analysed and fed to the the estimation model. The model outputs an early estimation for the output metrics, considering the optimisation phases for each platform.



Source: Elaborated by the author.

B.1 DAMICORE Approach

Chart 12 presents the tools used by our first approach, and how they relate to the model presented in Figure 48. A clusterisation tool named DAMICORE is used to find a relation between the KUT and kernels that make part of a reference set. Depending on how the KUT kernel is clustered among the reference kernels, a decision is taken on whether the KUT is FPGA or GPU suitable. Our tool `mdamicore2` expands DAMICORE by allowing multiple representations to be considered at once, such as data-flow graph, source code, etc. We use LLVM to generate these different representations.

Chart 12 – Tools used by our first approach, related to the abstract model previously presented.

From Figure 48	First approach
Estimation model	`mdamicore2`
Code analyser	LLVM standard passes
FPGA code optimiser	None
FPGA synthesis & execution	Altera SDK for OpenCL
GPU code optimiser	None
GPU compilation & execution	NVIDIA OpenCL SDK

Source: Elaborated by the author.

This approach can be summarised in the following steps:

- **Training phase**

  1. Considering k kernels available from an initial reference set, prepare the r different representations for each;
  2. Execute `mdamicore2` considering all kernels and representations;
  3. Analyse the generated clusterisation, and using a suitability ranking based on energy efficiency, remove the kernels that less contribute to each formed cluster (e.g. a kernel marked as GPU-suitable that was included in a heavily FPGA-suitable cluster);
  4. Repeat from step 2 until a minimum number of kernels is reached.

- **Use phase**

  1. Prepare the r representations for the KUT;
  2. Create an input set for `mdamicore2` containing the representations from the KUT, and also from the reference set generated in the training phase;
  3. Execute `mdamicore2`;
  4. Analyse where the KUT was positioned (i.e. on which cluster and/or close to which reference kernels), and use a selection criterion to decide if the KUT is FPGA-suitable or not.

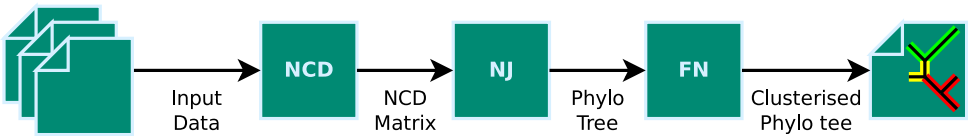
The next section presents the `mdamicore2` tool used for data mining. Then, subsection B.1.2 presents the reference set generation, subsection B.1.3 presents the decision making criterion used, subsection B.1.4 presents the quality metrics used to evaluate our estimation, subsection B.1.5 presents the initial kernel set, and subsection B.1.6 presents early results.

B.1.1 The `mdamicore2` Tool

The Data Mining of Code Repositories (DAMICORE) (SANCHES; CARDOSO; DELBEM, 2011) is a clustering algorithm based on finding similarities by compression. The use of compression for similarity analysis implies that any representation of data may be used, from source codes to features extracted from them, in text or binary form.

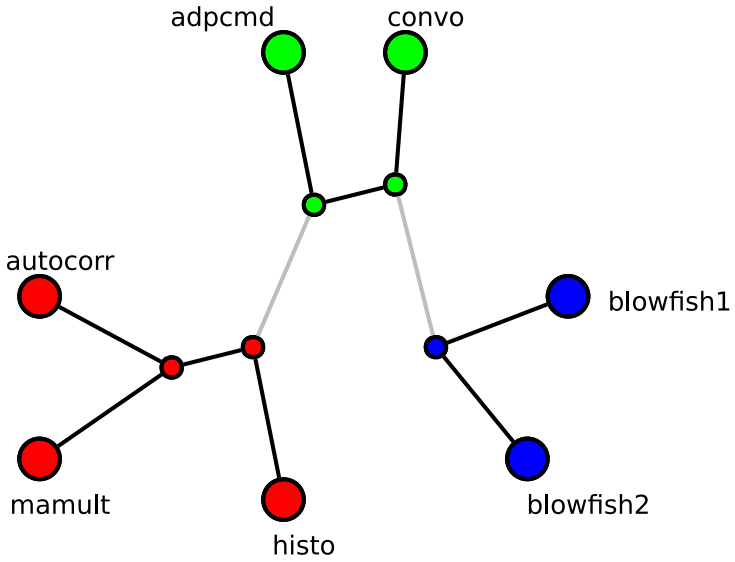
This methodology combines three techniques: Normalised Compression Distance (NCD), Neighbour Joining (NJ) and Fast Newman (FN), as shown in Figure 49. In short terms, DAMICORE receives as input a set of data of the same type, creates a matrix with NCD values from all pair-wise combinations, uses this matrix to generate a phylogenetic tree using Neighbour Joining and finally detects clusters using Fast Newman. Figure 50 presents a typical DAMICORE output in visual format.

Figure 49 – DAMICORE flow.



Source: Elaborated by the author.

Figure 50 – Visual representation of a DAMICORE output. Each labelled node is a kernel.



Source: Elaborated by the author.

Sumoyama (2016) developed a similar approach as ours, however a different HLS compiler was considered and GPUs were not targetted. Additionally, the kernel source code was used as sole input representation for DAMICORE in his approach. This has some drawbacks, for example code styling might heavily affect the results.

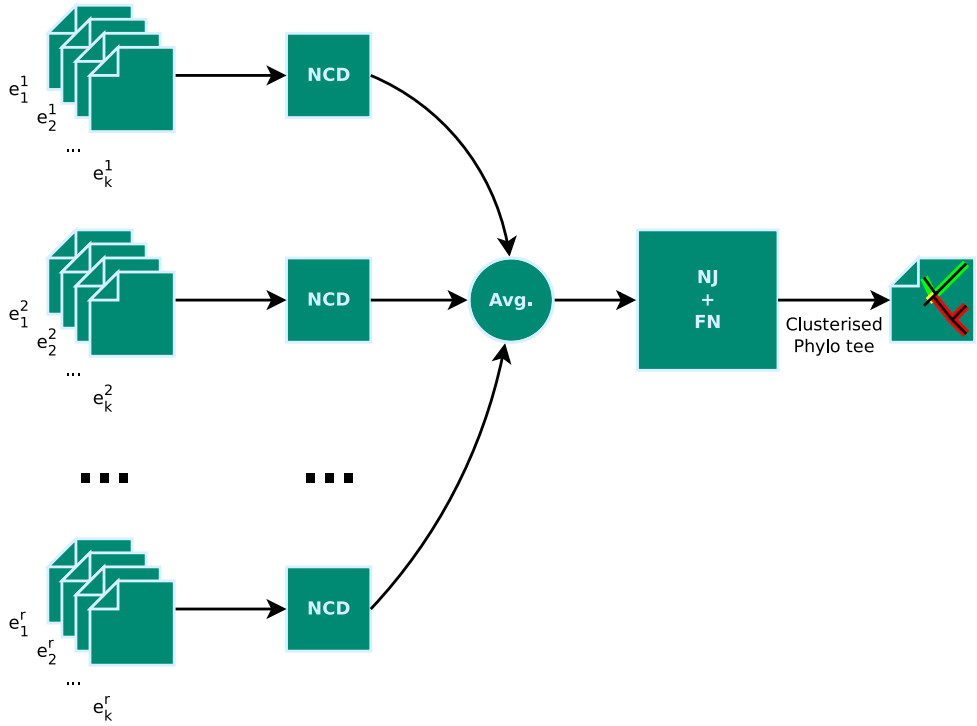
Our developed tool, `mdamicore2`, supports multiple representations as input simultaneously. This has the advantage of aggregating information about the kernel from

different perspectives. Consider an input data set of k kernels $(e_1, e_2, ..., e_k)$, each having r different representations of its information $(e_i^1, e_i^2, ..., e_i^r)$. For each representation r, `mdamicore2` executes the compression and NCD matrix generation. An NCD matrix for each type of representation is generated, and `mdamicore2` merges them by calculating the average matrix NCD_{avg}:

$$NCD_{avg} = \frac{\sum_1^r NCD_i}{r} \tag{B.1}$$

The resultant NCD_{avg} matrix is then delivered to the next steps of DAMICORE (NJ and FN). Figure 51 presents such approach:

Figure 51 – Overview of `mdamicore2` using average NCD matrix calculation.



Source: Elaborated by the author.

B.1.2 Reference Set Generation

A reference set with a good coverage on diverse code patterns and platform suitabilities is essential for quality estimations. Considering an initial kernel set (e.g. kernels from Rodinia, SHOC), all applications are first compiled and executed on both platforms. Then, the energy efficiency of each architecture is used as criterion for keeping or removing kernels, until a set with enough coverage is reached.

Consider $e = p \times t$ as the consumed energy in terms of average power p and execution time t. We propose the anti-suitability values s_i ($i \in \{\text{GPU}, \text{FPGA}\}$), as the metric to

be used during reference set generation. The anti-suitability for platform i is calculated as follows:

$$s_i = \frac{e_i}{\max(e_{\text{GPU}}, e_{\text{FPGA}})} \tag{B.2}$$

The anti-suitability value ranges from 0 to 1. A value of $s_i = 1$ indicates that architecture i is less suitable than the counterpart for the associated kernel.

Our reference set generation method is derived from the "Best Per-Group Coverage" approach as presented by Martins (2015), which attempts to maximise the reference set quality on a cluster basis instead of a global optimisation. In our case, a reference set with ideal coverage would be composed of clusters that define different patterns of kernels sharing common anti-suitability values. For the following method explanation, let S be the list of candidate kernels to be removed from reference set; and let p and $\bar{p}$ be a notation to represent both platforms here considered, where $\bar{p}$ is the counterpart of p (e.g. if $p = \text{FPGA}$ then $\bar{p} = \text{GPU}$, or if $p = \text{GPU}$ then $\bar{p} = \text{FPGA}$). Our method is then defined as:

1. The initial kernel set is clusterised using `mdamicore2`;
2. Then for every cluster:
   a) Calculate s_p^k and $s_{\bar{p}}^k$ for all kernels k within a cluster and decide the most suitable platform (if $s_p^k < s_{\bar{p}}^k$, k is suitable for p or vice-versa);
   b) Label cluster as p-dominant if there are more kernels suitable for p than $\bar{p}$ or vice-versa. If the number of dominant kernels is the same, find weak dominancy:
      - If $\sum_k s_p^k < \sum_k s_{\bar{p}}^k$, cluster is p-(weak)-dominant or vice-versa;
   c) If cluster is p-dominant (or weak), append to S the kernel with the smallest $s_{\bar{p}}$. If more than one kernel fits this criteria, append to S the one containing the largest s_p value;
3. Check if there are kernels in S that are p-suitable but clusterised in a $\bar{p}$-dominant cluster;
   a) If positive, remove the one with the smallest s_p;
   b) If negative, remove the kernel with the biggest $s_{\bar{p}}$;
4. Repeat all above steps until a minimum number of kernels is reached.

B.1.3 Decision Making

The generated reference set may then be used along with a KUT kernel to estimate anti-suitability values and to decide which platform is best. Consider $\hat{s}_i$ as being our estimated value for s_i, C as the set containing all kernels of the cluster where the KUT was included, and d_j as the NCD value between the KUT and j kernel (i.e. the similarity distance between both kernels). Adapted from Sumoyama (2016), we implement three methods for estimating the anti-suitability values based on the KUT's positioning:

- **Cluster (c):** average value from all kernels within the cluster:

$$\hat{s}_i = \frac{\sum_{k \in C} s_i^k}{|C|} \tag{B.3}$$

- **Relative (r):** the s_i values from the closest relative are used:

$$\hat{s}_i = s_i^k | k \in C, d_k = \min_{m \in C}(d_m) \tag{B.4}$$

- **Cluster + Relatives (cr):** weighted average value from all kernels within the cluster. The NCD values are used to weight the average:

$$d'_k = \max_{m \in C}(d_m) - d_k \tag{B.5a}$$

$$\hat{s}_i = \frac{\sum_{k \in C} d'_k \cdot s_i^k}{\sum_{k \in C} d'_k} \tag{B.5b}$$

  where d'_k is a complement value of d_k considering the maximum d value among all kernels in the cluster. We use this value as weight instead of d_k, since greater distance implies in less influence. Thus, such value should weight less on the average value.

B.1.4 Quality Metric

We evaluate the quality of a generated reference set by using another set of kernels as KUT inputs. The values estimated by our model for these kernels are then compared with their actual values. Considering K as the set of validation KUT kernels, two metrics are defined:

- Let q_a be the quality metric for a given decision making approach a ($a \in \{\mathrm{c}, \mathrm{r}, \mathrm{cr}\}$):

$$q_a = \frac{|H_a|}{|K|} \tag{B.6}$$

  where H_a is the set of kernels h ($h \in K$) on which approach a correctly inferred the most suitable platform;

- Let err_i be the average anti-suitability error for platform i, defined as:

$$err_i = \frac{\sum_{k \in K} |s_i^k - \hat{s}_i^k|}{|K|} \tag{B.7}$$

where s_i^k is the true anti-suitability value derived from execution results for platform i and kernel k, and $\hat{s}_i^k$ is the inferred value by our model.

B.1.5 Initial Kernel Set

Our initial kernel set used to generate the reference and validation sets is composed of 50 kernels, where 45 of them were collected from three benchmarks: SHOC[1], Rodinia[2] and CHO[3] (NDU; NAVARIDAS; LUJÁN, 2015). The 5 remaining kernels are from an in-house adaptation of a Reed-Solomon Decoder[4].

We attempted to run all kernels on the following platforms:

- **FPGA:** BittWare S5PH-Q (Intel FPGA Stratix V);
- **GPU:** NVIDIA Quadro K620.

The following representations were generated for each kernel:

- **src:** kernel source code;
- **xml-full:** kernel XML description file. This file is used by `hostcodegen` to generate the OpenCL host code (see Appendix A for more information);
- **xml-norm:** normalised kernel XML description (kernel and variable names are omitted);
- **ir:** kernel intermediate representation generated by the LLVM compiler;
- **cfg-full-(bmp|ps|plain):** control-flow graph generated by the LLVM compiler in BMP, PostScript or pure-text formats;
- **cfg-norm-(bmp|ps|plain):** normalised control-flow graph generated by the LLVM compiler (only the graph structure is maintained) in BMP, PostScript or pure-text formats.

[1] SHOC kernels: bfs, fft, gemm, md, md5hash, reduction, spmv, stencil2d, scan.

[2] Rodinia kernels: hotspot, kmeans, lavamd, nn, nw1, nw2, pathfinder, srad, backprop1, backprop2, lud1, lud2, lud3, particlefilter1, particlefilter2, leukocyte1, leukocyte2, hotspot3d, hybridsort1, hybridsort2, hybridsort3, streamcluster, cfg, bptree.

[3] CHO kernels: aes_enc, aes_dec, gsm, jpeg, sha, dfsin, dfmul, dfdiv, adpcm, motion, blowfish, mips.

[4] Reed-Solomon kernels: rsd1, rsd2, rsd3, rsd4, rsdfull. These are also based on the implementation of Agarwal, Ng *et al.* (2010) as the ones used in Appendix C. However, the ones here used implement the task execution model instead.

B.1.6 Results

This section presents an early validation study for the proposed model. First, we present the execution results for the initial kernel set (which kernels executed, which did not, etc.). Then we use the kernels that successfully ran on both platforms to evaluate our model.

B.1.6.1 Execution Results for the Initial Kernel Set

Not every kernel from the initial set compiled or executed on both platforms. Reasons include:

- Some did not compile for the FPGA due to internal compiler errors, such as segmentation fault or HLS compiler error;
- Some did not fit on the FPGA due to excessive resource usage;
- Some did not execute in GPU due to unsupported data types (e.g. `long long`) or due to NDRange errors.

Chart 13 presents the compile and execution success for each kernel. For the ones that executed on both platforms, we calculated the anti-suitability values and selected the most suitable platform. We calculate the energy values by considering each platform's TDP[5] (i.e. $e = p \times t$, where p is the TDP and t the execution time). From all 50 kernels, 20 did not execute on FPGA or GPU (or both). From the 30 remaining, only 10 are FPGA-suitable.

B.1.6.2 Evaluation of Proposed Model

For this section, consider that $\mathbb{O}$ is the set containing all kernels that executed on both platforms, $\mathbb{K}$ is the set of KUTs used to evaluate our method, $\mathbb{I}$ is the initial set of kernels used for reference set generation and $\mathbb{F}$ is the final reference set achieved[6]. We then conducted six experiments, each with a variation on the set sizes and the number of input representations considered. Table 6 presents the experiments and the parameters considered for each.

Considering all experiments, 300 reference sets were generated. These were divided as follows:

- For each experiment, five sub-experiments are generated;

[5] Thermal's Design Power, or TDP. The TDP can be seen as a worst-case power drain for a platform.

[6] $\mathbb{K}, \mathbb{I}, \mathbb{F} \subset \mathbb{O}$; $\mathbb{F} \subset \mathbb{I}$; and $\mathbb{K} \cap \mathbb{I} = \emptyset$.

Chart 13 – Execution results for the initial kernel set.

Kernel	Exec. success		Best platform
	FPGA	GPU	
SHOC			
bfs	✓		—
fft		✓	—
gemm		✓	—
md	✓	✓	GPU
md5hash	✓	✓	GPU
reduction	✓	✓	GPU
spmv		✓	—
stencil2d			—
scan		✓	—
Rodinia			
hotspot		✓	—
kmeans	✓	✓	GPU
lavamd		✓	—
nn	✓	✓	GPU
nw1	✓	✓	GPU
nw2	✓	✓	GPU
pathfinder			—
srad	✓	✓	GPU
backprop1	✓	✓	GPU
backprop2	✓	✓	GPU
lud1	✓	✓	GPU
lud2	✓	✓	GPU
lud3		✓	—
particlefilter1		✓	—
particlefilter2		✓	—
leukocyte1	✓	✓	GPU
leukocyte2	✓	✓	GPU
hotspot3D	✓	✓	GPU
hybridsort1	✓	✓	GPU
hybridsort2	✓	✓	GPU
hybridsort3	✓	✓	GPU
streamcluster	✓	✓	GPU
cfd	✓	✓	GPU
bptree		✓	—

Kernel	Exec. success		Best platform
	FPGA	GPU	
CHO			
aes_enc	✓	✓	FPGA
aes_dec	✓	✓	FPGA
gsm	✓	✓	FPGA
jpeg		✓	—
sha	✓		—
dfsin	✓		—
dfmul	✓		—
dfdiv	✓		—
adpcm	✓	✓	FPGA
motion			—
blowfish	✓		—
mips	✓	✓	FPGA
Personal			
rsd1	✓	✓	FPGA
rsd2	✓	✓	FPGA
rsd3	✓	✓	FPGA
rsd4	✓	✓	FPGA
rsdfull	✓	✓	FPGA

Source: Research data.

Table 6 – Experimental setup for the `mdamicore2` approach.

Experiment #	$\lvert\mathbb{K}\rvert$	$\lvert\mathbb{I}\rvert$	$\lvert\mathbb{F}\rvert$	# of representations
1	10	20	14	2
2	12	18	12	2
3	14	16	12	2
4	10	20	14	3
5	12	18	12	3
6	14	16	12	3

Source: Research data.

- For each sub-experiment, the kernels in $\mathbb{O}$ are randomly split between $\mathbb{K}$ and $\mathbb{I}$ according to the set sizes defined by the experiment. The available input representation types are also randomly sampled according to the amount defined by the experiment;
- Then, for each sub-experiment. Ten reference sets $\mathbb{F}$ are generated and validated using the kernels in $\mathbb{K}$.

Table 7 and Table 8 present the best, worst and average quality metrics in each experiment. In some cases the KUT was placed on an isolated cluster. When this occurred, our model was not able to take a decision using cluster-based approaches (i.e. `c` and `cr`) and thus they are not considered in the experiments. The `r` approach, however, does not have this issue.

From the best and worst cases results, first experiment has the smallest accumulated error using only 2 representations (`xml-full` and `cfg-norm-ps`). Experiment 5, however, has almost the same accumulated error and 100% of hit rate. Experiment 1 has also the least worst cases among all. In average, all experiments are similar, with a still noticeable average error ranging from 0.150 to 0.350. However, qualities range in average from 70% to 88% hit rates, an improved rate when compared to simple random pick of platform.

Figure 52 presents the tree generated for the best case of experiment 5. In this case, all FPGA kernels were clusterised together.

The representations most frequently present in best and worst cases were:

- **Most frequent representation in best cases:** `xml-norm`;
- **Most frequent representation combination in best cases:**
  - **2 representations:** `xml-full`, `xml-norm`;
  - **3 representations:** `xml-norm`, `ir`, `cfg-full-bmp`;

Table 7 – Best and worst execution cases for each experiment.

Best									
Experiment	Qualities			err_{FPGA}			err_{GPU}		
#	q_c	q_r	q_{cr}	c	r	cr	c	r	cr
1	1.000	0.900	1.000	0.085	0.179	0.085	0.011	0.093	0.010
2	1.000	1.000	1.000	0.122	0.115	0.122	0.105	0.107	0.104
3	0.929	1.000	0.929	0.158	0.076	0.152	0.123	0.085	0.120
4	1.000	1.000	1.000	0.119	0.131	0.119	0.051	0.046	0.051
5	1.000	1.000	1.000	0.087	0.172	0.089	0.045	0.046	0.045
6	0.929	1.000	0.929	0.076	0.128	0.076	0.106	0.077	0.108
Worst									
Experiment	Qualities			err_{FPGA}			err_{GPU}		
#	q_c	q_r	q_{cr}	c	r	cr	c	r	cr
1	0.600	0.600	0.600	0.374	0.362	0.390	0.310	0.303	0.310
2	0.583	0.583	0.583	0.501	0.536	0.501	0.328	0.328	0.328
3	0.500	0.500	0.500	0.470	0.574	0.470	0.346	0.346	0.346
4	0.600	0.500	0.600	0.374	0.494	0.390	0.343	0.486	0.367
5	0.500	0.500	0.500	0.533	0.581	0.562	0.363	0.363	0.363
6	0.500	0.500	0.500	0.487	0.569	0.486	0.398	0.398	0.398

Source: Research data.

- **Most frequent representation in worst cases:** `cfg-full-bmp`;
- **Most frequent representation combination in worst cases:**
  - **2 representations:** `cfg-full-bmp`, `cfg-full-ps`;
  - **3 representations:** `cfg-norm-bmp`, `cfg-norm-ps`, `cfg-full-bmp`.

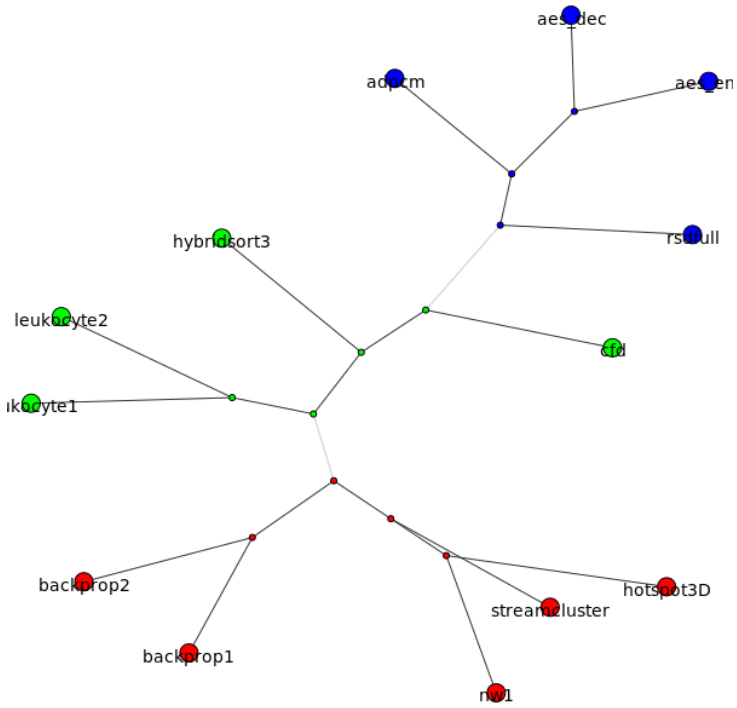
The XML description files were frequently present in the best cases, indicating its potential importance as data mining input. The LLVM IR code and control-flow graph (preferrably in image format) were also frequently present. It is possible to note that the worst cases frequently included representations that describe the same information, but on a different format (e.g. control-flow graph in text and image format both being considered at the same time). This indicates the importance of having a diverse set of input representations.

Table 8 – Average values for each experiment.

Average									
Experiment	Qualities			err_{FPGA}			err_{GPU}		
#	q_c	q_r	q_{cr}	c	r	cr	c	r	cr
1	0.812	0.801	0.828	0.253	0.286	0.255	0.160	0.169	0.156
2	0.723	0.734	0.728	0.311	0.353	0.319	0.274	0.240	0.266
3	0.810	0.809	0.818	0.244	0.251	0.241	0.201	0.205	0.193
4	0.828	0.844	0.830	0.226	0.241	0.225	0.182	0.160	0.180
5	0.803	0.796	0.817	0.240	0.279	0.236	0.219	0.214	0.214
6	0.780	0.809	0.794	0.251	0.249	0.250	0.222	0.210	0.219
Average w/o first quartile									
Experiment	Qualities			err_{FPGA}			err_{GPU}		
#	q_c	q_r	q_{cr}	c	r	cr	c	r	cr
1	0.864	0.837	0.878	0.219	0.257	0.220	0.133	0.158	0.130
2	0.782	0.773	0.787	0.270	0.317	0.269	0.245	0.224	0.237
3	0.871	0.847	0.871	0.203	0.219	0.199	0.161	0.195	0.156
4	0.874	0.891	0.874	0.200	0.211	0.200	0.149	0.129	0.148
5	0.864	0.846	0.879	0.195	0.243	0.191	0.192	0.187	0.186
6	0.824	0.858	0.839	0.223	0.221	0.221	0.195	0.182	0.193

Source: Research data.

Figure 52 – Generated phylogenetic tree for best case in experiment 5.



Source: Research data.

B.2 Neural Network Approach

In the second approach, we used Artificial Neural Network (ANN) models to estimate the output metrics. Chart 14 presents the tools used and how they relate to the model presented in Figure 48. Given a KUT kernel, numerical features are extracted using our code analyser named OpCount. These features are fed to the neural network model, which estimates the energy consumption for the KUT.

Chart 14 – Tools used by our second approach, related to the abstract model previously presented.

From Figure 48	Second approach
Estimation model	Artificial neural network
Code analyser	OpCount (LLVM-based)
FPGA code optimiser	None
FPGA synthesis & execution	Altera SDK for OpenCL
GPU code optimiser	None
GPU compilation & execution	NVIDIA OpenCL SDK

Source: Elaborated by the author.

The next section formulates the inputs and expected outputs of the model, subsection B.2.2 describes the neural network setups used, and subsection B.2.3 presents the

experimental validation results.

B.2.1 Formulation and Methodology

The ANN models perform numerical transformations, i.e. both inputs and outputs are numbers. Thus it is still necessary to formulate how the OpenCL kernels are converted from source code to ANN input variables and which output metrics we are expecting for the ANN to estimate.

B.2.1.1 ANN Inputs

We formulate the ANN inputs as being code features statically extractable from an OpenCL kernel. First we use LLVM to compile from OpenCL to LLVM IR. Then a custom LLVM pass named OpCount extracts the code features as presented in Chart 15.

Chart 15 – Extracted code features using OpCount.

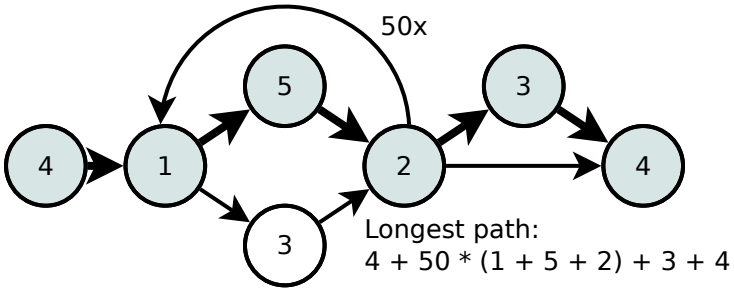
Feature code	Feature name	Longest path criterion
`lp`	Longest path	# of instructions
`noi`	Naive operational intensity	# of instructions
`nmi`	Naive memory intensity	# of bytes transferred
`fpops`	Floating-point operations	# of FPOps
`bars`	Number of barriers	# of barriers
`tc`	Maximum trip count	—
`ldep`	Deepest loop depth	—

Source: Perina and Bonato (2018).

Feature `lp` counts the number of IR instructions along the longest path of the control-flow graph (CFG) for the OpenCL code. Due to presence of loop back-edges in the CFG making them directed cyclic graphs, finding the longest path is NP-hard. To overcome this issue, we remove all loop back-edges and we compensate by multiplying the affected CFG nodes with their respective loop's trip count. Figure 53 presents an example of a CFG and its longest path. This metric can be interpreted as the worst possible case of execution, where loops are fully executed and the longest blocks are always taken from conditionals.

Both `noi` and `nmi` metrics are composed by counting the number of bytes transferred by loads and stores instructions and dividing by the longest path of the CFG. In the first metric, bytes are counted in the same path as the `lp` metric while the latter considers the longest path where the most amount of bytes has been transferred. Both metrics are based on the operational intensity concept as presented by Williams, Waterman and Patterson (2009) in their Roofline model. The naive characteristic comes from the fact that no input data is used to infer the code's execution path.

Figure 53 – Example of control-flow graph and its longest path in thicker edges. The weight of a node describes the amount of contained instructions. As an example, it is assumed a loop trip count of 50.



Source: Perina and Bonato (2018).

The `fpops` metric counts the number of instructions where at least one operand is of floating-point type, while the `bars` metric counts the number of OpenCL barriers. Both counts consider worst-case scenario (e.g. for `bars`, the path with the most amount of barriers).

Finally, `tc` exposes the maximum trip count within a CFG, while `ldep` exposes the maximum loop depth in this graph.

B.2.1.2 ANN Outputs

Our models estimate the following outputs:

- **Energy Consumption:** the amount of energy consumed by a kernel. Currently this value is acquired by multiplying the execution time by the architecture's TDP;
- **Class:** after calculating energy consumption, the kernel can be assigned to a class (i.e. FPGA or GPU) by considering the smallest consumption.

B.2.2 Neural Network Experimental Setup

Using MathWorks MATLAB R2015a, the following neural networks were used: learning vector quantisation (`lvq`), multi-layer perceptron (`mlp`) and radial basis function (`rbf`). In each network, several parameters were varied in order to explore different topologies and their performances. A single combination of such parameters is henceforth called setup. Chart 16 presents the explored setup parameters for all networks.

Since `lvq` has a faster training than the other networks, we first explored using `lvq` with a broad set of feature combinations. Then, the best combinations were selected to be used with the other networks.

The cross-validation method with random subset sampling was used on all networks, where 100 trainings were performed for each possible setup. Each training/valida-

Chart 16 – Neural networks setup parameters.

LVQ	
Parameter	Possible values
No. of neurons	2, 4 and 8
Input variables sets	All features, (`lp`), (`lp`, `noi`), (`lp`, `nmi`), (`lp`, `fpops`, `bars`), (`lp`, `noi`, `nmi`, `fpops`, `bars`), (`lp`, `noi`, `nmi`, `fpops`, `bars`, `ldep`), (`noi`, `nmi`, `fpops`, `bars`, `ldep`), (`fpops`, `bars`)
Output metric	Class assignment
MLP	
Parameter	Possible values
Number of hidden layers	1, 2 and 3
Hidden layers topology	(5), (10), (50), (5, 5), (5, 10), (5, 50), (10, 10), (10, 50), (50, 50), (5, 5, 5), (5, 5, 10), (5, 5, 50), (5, 10, 10), (5, 10, 50), (5, 50, 50), (10, 10, 10), (10, 10, 50), (10, 50, 50), (50, 50, 50)
Input variables sets	All features and also the most accurate combinations from LVQ
Output metric	Energy consumption and class assignment
RBF	
Parameter	Possible values
Spread	0.02, 0.03, 0.04, 0.05, 0.06 and 0.07
Input variables sets	All features and also the most accurate combinations from LVQ
Output metric	Energy consumption and class assignment

Source: Perina and Bonato (2018).

tion phase produces results that are analysed by a performance metric. For the continuous energy consumption, Root-Mean-Squared Error (RMSE) was used:

$$RMSE = \sqrt{\frac{\sum_{k \in K} (\hat{y}_k - y_k)^2}{|K|}} \tag{B.8}$$

where K is the set of kernels used for validation, $\hat{y}_k$ is the estimated value by our model and y_k is the actual value. For the discrete class assignment metric (i.e. $\hat{y}_k, y_k \in \{\mathrm{FPGA}, \mathrm{GPU}\}$), hit rate was used:

$$HIT = \frac{|H|}{|K|} \tag{B.9a}$$

$$H = \{k \in K \mid \hat{y}_k = y_k\} \tag{B.9b}$$

where H is the set containing all kernels where the correct platform was inferred. For example, a network with a 0.9 hit rate implies that it was able to correctly infer the most suitable accelerator for 90% of the validation subset.

After all 100 trainings, the best, worst and average performance metrics were calculated for each possible setup. All input variables and output metrics were normalised prior to training and validation.

The same kernel set from the previous approach was used. Please refer to subsection B.1.5 for detailed information about the kernels and platforms.

B.2.3 Experimental Results

Table 9 presents performance results for all networks. In `lvq`, results for all extracted features and two other setups with the best average performance are presented, while for `mlp` and `rbf` only the best setup is presented.

For `lvq`, the best average performance was for the setup (`lp`, `nmi`) with 4 neurons, reaching almost 85%. For `mlp`, the estimation error for energy is significant as pointed by the RMSE: the best setup has an average error of 16563.9 and 5362.9 for GPU and FPGA respectively for an unnormalised interval of [2.0;62507.4] for GPU and [1.7;20239.1] for FPGA, all in kilojoules (kJ). For `rbf`, interestingly the best performance was found when using all input variables, though several other setups for this network had almost the same performance. Considering energy consumption, not only did `rbf` perform slightly better (average error of 14376.2kJ and 4654.6kJ for GPU and FPGA respectively) but the worst RMSE was also smaller.

For both `lvq` and `mlp`, the input variables combination (`lp`, `nmi`) was present in almost all best results. For class assignment in `mlp` and `rbf`, the combination (`lp`, `noi`, `nmi`, `fpops`, `bars`) performed better.

A likely cause for the significant energy estimation error is the size of training and cross-validation subsets, not having sufficient coverage for an ideal estimation. However, the `lvq` network was able to correctly infer the most suitable platform for an average of almost 85%, outperforming all other approaches. Such result could be further improved by increasing the number of samples for training and validation.

Table 9 – Performance results for all networks.

LVQ					
			HIT		
Output	Input variables	Neurons	Worst	Best	Avg.
Class	All	4	0.400	1.000	0.764
	(`lp`, `nmi`)	4	0.400	1.000	0.849
	(`lp`, `noi`, `nmi`, `fpops`, `bars`)	8	0.400	1.000	0.826
MLP					
			RMSE / HIT		
Output	Input variables	Neurons	Worst	Best	Avg.
Energy	(`lp`, `nmi`)	(5, 5)	0.664	0.063	0.265
Class	(`lp`, `noi`, `nmi`, `fpops`, `bars`)	(5)	0.200	1.000	0.798
RBF					
			RMSE / HIT		
Output	Input variables	Spread	Worst	Best	Avg.
Energy	All	0.03	0.385	0.032	0.230
Class	(`lp`, `noi`, `nmi`, `fpops`, `bars`)	0.07	0.400	1.000	0.716

Source: Perina and Bonato (2018).

B.3 Final Remarks

This appendix presented the early approaches in regard to estimation of metrics involving OpenCL kernels on FPGA and GPU. Two approaches were presented to estimate the best platform and the energy consumption of a given test kernel. The first approach used a data mining clusterisation tool and was able to reach an average hit rate of 70% to 88% when estimating the most suitable platform for a kernel. The second approach was modelled around neural networks, and was able to reach nearly 85% of hit rate.

Both models, however, have significant errors when estimating the energy consumption values. One of the reasons is the initial kernel set size used for generating the reference sets and training, which was still small. Furthermore, the kernels designed for GPUs (Rodinia and SHOC) had poor performance on FPGAs, and conversely the kernels from CHO performed bad on GPUs. We then proceeded on finding more kernels and also to study the proposed code optimiser as depicted in Figure 48. The findings of this posterior phase is presented on Appendix C, which indicated that a code optimiser on the OpenCL level would be too complex.

APPENDIX

C

COMPARATIVE ANALYSIS OF OPENCL KERNELS IN FPGA AND GPU

In this appendix, we present a comparison analysis of OpenCL performance portability. We executed a set of OpenCL kernels with varying degree of HLS optimisation efforts, and we assessed the FPGA performance standpoint of these kernels in comparison to GPU. This analysis was motivated after our results using machine learning models, as presented in Appendix B. When gathering a set of OpenCL kernels that would compose our machine learning training and validation sets, we noticed that the kernels were performing really bad on the platforms they were not designed for.

All kernels and analyses can be found in our project repository[1]. In the following section, we present the two OpenCL execution models and how they relate to FPGAs and GPUs.

C.1 OpenCL Execution Models

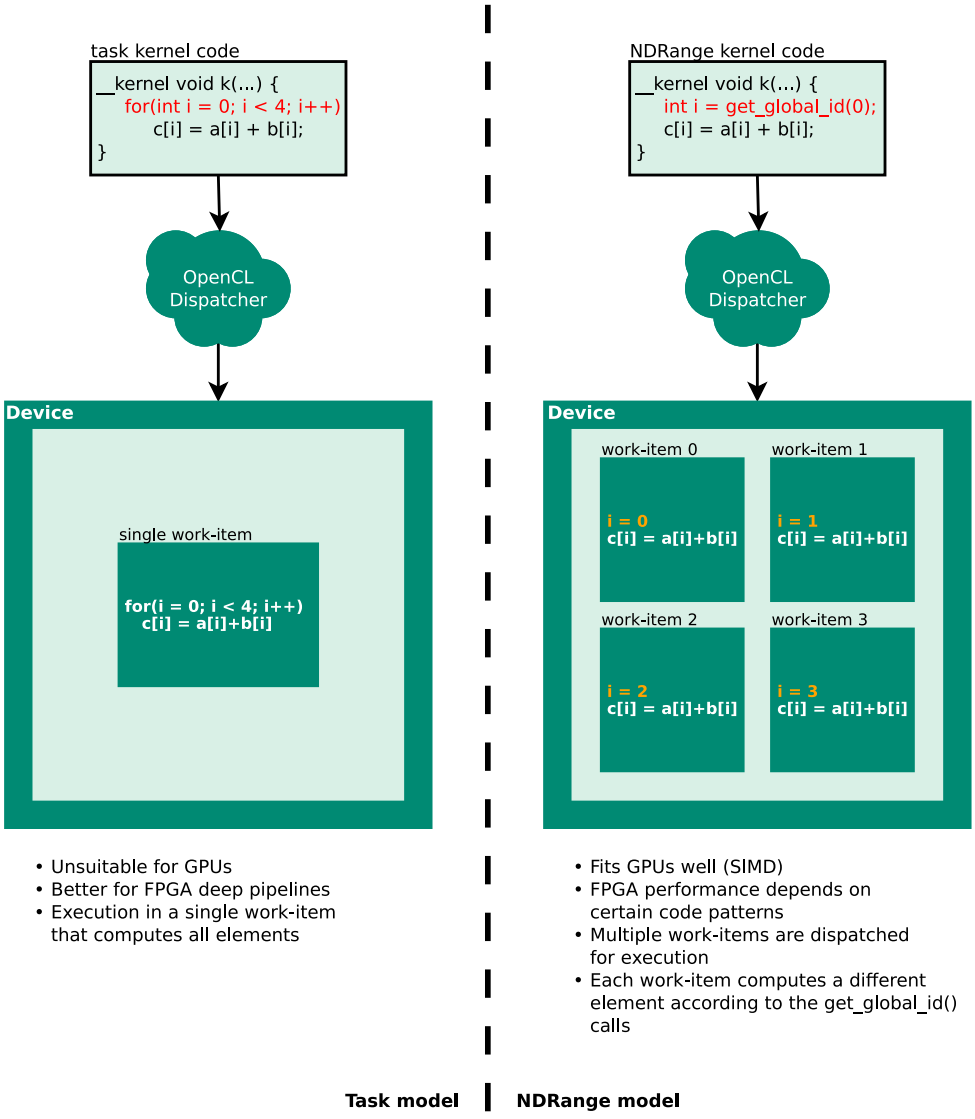
An OpenCL application is divided in two parts: the device and the host codes. The device code is the OpenCL kernel, a function written using C-alike semantics that executes on the accelerator. The host code is a C/C++ application that uses the OpenCL API and is responsible for: allocating resources to access the device; managing the communication from/to the device; dispatching kernels for execution; etc. Usually, the OpenCL host is a system running Linux/Windows and the OpenCL device is an accelerator attached to the host using a high-speed communication bus (e.g. PCI Express).

Two execution models are available when designing OpenCL kernels: task and NDRange. When using the task model, the kernel executes on the accelerator using a single compute core, similar to a sequential software execution in a single CPU core.

[1] https://github.com/comododragon/opencl-4all

When using the NDRange model, the OpenCL framework instantiates several work-items to be executed in a SIMD fashion, each processing a different set of input/output data. Then, these work-items are scheduled for execution in one or more compute units within an accelerator. Figure 54 compares the execution mapping of a simple vector add when using task or NDRange models.

Figure 54 – Comparison between the task and NDRange models for a simple vector add.



Source: Elaborated by the author.

GPUs are naturally suited for the NDRange model and therefore most — if not all — GPU kernels available are designed in this format. FPGAs support both task and NDRange, however the suitability of each execution model is dependent on the application. The task model implemented on AOCL allows the exploration of deep pipeline parallelism, which is very suitable for FPGAs. However, in the case where the compiler is not able to implement an efficient pipeline when using the task model, the NDRange might still be benefitial for FPGAs. The reason is that NDRange employs dynamic pipelining (at

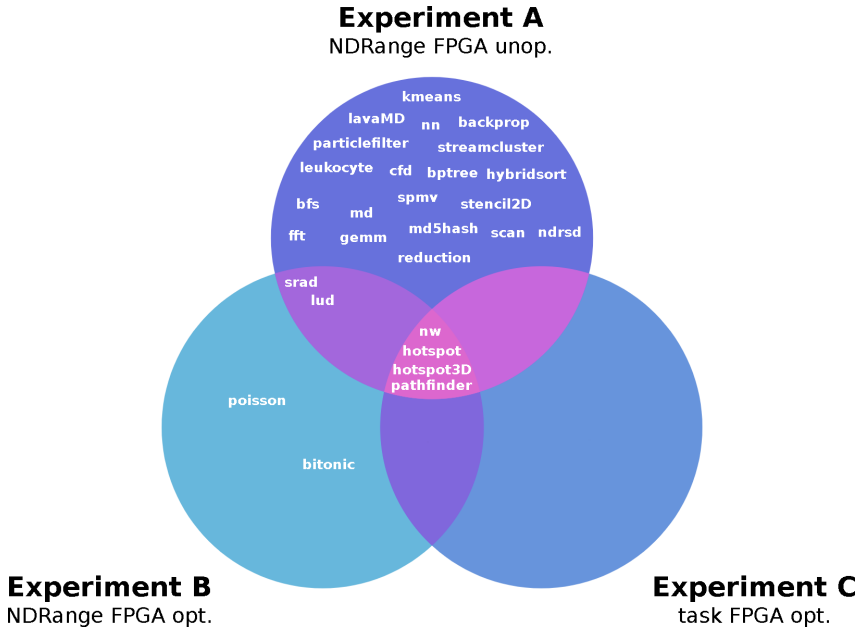
work-item level) that can achieve better results than the statically pipelined case of task model (ZOHOURI, 2018).

C.2 Kernel Set

The performance results of the kernels used in our machine learning models have shown that all OpenCL kernels designed for one architecture performed poorly on the one that they were not designed for. This is expected, as pointed by Zohouri (2018), Muslim *et al.* (2017) and Weller *et al.* (2017). As a consequence, our intention was to attach a model to our framework that would optimise NDRange kernels when targetting FPGAs. But, prior to developing such model, we decided to evaluate the performance standpoint of FPGAs when considering NDRange kernels, and also how much code optimisation would be necessary to achieve competitive performance against GPUs.

To this end, we gathered 56 kernels from varied sources and compared the performance and estimated energy consumption between FPGAs and GPUs. We divide these kernels in three classes, each considering a different level of optimisation effort towards HLS. There are similar kernels across classes, and these can be used to quantify the improvements proportional to the amount of HLS optimisations applied. Figure 55 presents a Venn diagram of the applications included in our kernel set, and how they spread across classes.

Figure 55 – Venn diagram of the kernels collected separated in three classes. Each class represents a different level of FPGA optimisation effort and execution model. Kernel variants (e.g. nw(1), nw(2)) are grouped.



Source: Elaborated by the author.

Some kernels were slightly modified to facilitate integration with our testbench and

with the FPGA vendor that we used. Similarly, the optimisations not supported by our vendor were removed. Please refer to the provided repository for a complete description of our modifications and the optimisations implemented in each kernel.

C.2.1 Class I: FPGA-unoptimised NDRange Kernels

The first class includes kernels that are of type NDRange, without any optimisation towards FPGA. There are 38 kernels in total: 24 from Rodinia, 9 from SHOC, and 5 kernels from a Reed-Solomon application. The Reed-Solomon kernels are based on an implementation of Agarwal, Ng *et al.* (2010), which we converted to OpenCL and manually mapped to the NDRange model.

Chart 17 presents the kernels for this class. Since all are NDRange with no FPGA optimisations, this class exemplifies the case where one simply tries to execute a kernel created for GPU in an FPGA.

C.2.2 Class II: FPGA-optimised NDRange Kernels

The second class includes NDRange kernels that were analysed and optimised for FPGA, and it is composed of 9 kernels from Zohouri (2018), 2 from Weller *et al.* (2017) and 3 from Muslim *et al.* (2017). The kernels added from Zohouri (2018) are optimised variants of some Rodinia kernels present in the first class.

Chart 18 presents the 14 kernels gathered. This class exemplifies the cases where one tried to optimise NDRange kernels for FPGA without changing its execution model.

C.2.3 Class III: FPGA-optimised Task Kernels

According to Zohouri *et al.* (2016), the task execution model is more suitable for FPGAs, although for some applications the NDRange model might still suit better. The task model usually reflects in deeper pipelines, which is adequate for FPGAs.

In this class, we include kernels specifically written for FPGA execution using the task model. These kernels are heavily optimised variants of Rodinia applications present in the previous classes. The codes went through a significant rewrite, since the whole execution model was swapped.

Chart 19 presents the 4 kernels included. This class exemplifies the cases where one heavily optimised an OpenCL kernel for FPGA, including the swap from NDRange to task model.

Chart 17 – FPGA-unoptimised NDRange kernels.

Kernel	Data size	Source
Hotspot	500×500 data points	Rodinia
K-means	30000 points, 34 features	Rodinia
LavaMD	10 boxes1d, 128 threads/block, 1000 blocks	Rodinia
NN	42764 records	Rodinia
NW(1)	2048×2048 data points	Rodinia
NW(2)	2048×2048 data points	Rodinia
Pathfinder	100 rows, 10000 cols	Rodinia
SRAD	502×458 image, 100 iter., 0.5 lambda	Rodinia
Backprop(1)	65536 nodes	Rodinia
Backprop(2)	65536 nodes	Rodinia
LUD(1)	1024×1024 matrix	Rodinia
LUD(2)	1024×1024 matrix	Rodinia
LUD(3)	1024×1024 matrix	Rodinia
Leukocyte(1)	175×596 data points	Rodinia
Leukocyte(2)	219×640 data points	Rodinia
Hybridsort(1)	1000000 elements	Rodinia
Hybridsort(2)	1000000 elements	Rodinia
Hybridsort(3)	1000000 elements	Rodinia
Hotspot3D	$512 \times 512 \times 8$ data points, 3 iterations	Rodinia
CFD	485760 variables, 97152 areas and step factors	Rodinia
BPTree	7874 nodes, 1000000 records	Rodinia
Particlefilter(1)	128×128 image, 10 frames, 40000 particles	Rodinia
Particlefilter(2)	128×128 image, 10 frames, 40000 particles	Rodinia
Streamcluster	65536 points, 64 dimensions	Rodinia
BFS	1000 nodes, max. degree 3	SHOC
FFT	65536 points	SHOC
GEMM	128×128 matrices	SHOC
MD	12288 atoms	SHOC
MD5Hash	10000000 keyspace size, size of byte 7, 10 values per byte	SHOC
Reduction	131072 elements	SHOC
SpMV	Sparse matrix with 1024 rows	SHOC
Stencil2D	271392 data points	SHOC
Scan	262144 data points	SHOC
NDRSD(1)	255 blocks of 255 bytes, where 32 are check symbols	Personal
NDRSD(2)	255 blocks of 255 bytes, where 32 are check symbols	Personal
NDRSD(3)	255 blocks of 255 bytes, where 32 are check symbols	Personal
NDRSD(4)	255 blocks of 255 bytes, where 32 are check symbols	Personal
NDRSDFull	255 blocks of 255 bytes, where 32 are check symbols	Personal

Source: Elaborated by the author.

Chart 18 – FPGA-optimised NDRange kernels.

Kernel	Data size	Source
Hotspot	500×500 data points	Rodinia, Zohouri (2018)
NW(1)	2048×2048 data points	Rodinia, Zohouri (2018)
NW(2)	2048×2048 data points	Rodinia, Zohouri (2018)
Pathfinder	100 rows, 10000 cols	Rodinia, Zohouri (2018)
SRAD	502×458 image, 100 iter., 0.5 lambda	Rodinia, Zohouri (2018)
LUD(1)	1024×1024 matrix	Rodinia, Zohouri (2018)
LUD(2)	1024×1024 matrix	Rodinia, Zohouri (2018)
LUD(3)	1024×1024 matrix	Rodinia, Zohouri (2018)
Hotspot3D	$512 \times 512 \times 8$ data points, 3 iterations	Rodinia, Zohouri (2018)
Poisson(1)	2 `float16` vectors with 65536 elements	Weller *et al.* (2017)
Poisson(2)	2 `float16` vectors with 65536 elements	Weller *et al.* (2017)
Bitonic(1)	4096 elements	Muslim *et al.* (2017)
Bitonic(2)	4096 elements, stride of 2048	Muslim *et al.* (2017)
Bitonic(3)	4096 elements, stride of 32	Muslim *et al.* (2017)

Source: Elaborated by the author.

Chart 19 – FPGA-optimised task kernels.

Kernel	Data size	Source
Hotspot	500×500 data points	Rodinia, Zohouri (2018)
NW	2048×2048 data points	Rodinia, Zohouri (2018)
Pathfinder	100 rows, 10000 cols	Rodinia, Zohouri (2018)
Hotspot3D	$512 \times 512 \times 8$ data points, 3 iterations	Rodinia, Zohouri (2018)

Source: Elaborated by the author.

C.3 Evaluation Setup

In this section, we define the evaluation setup used, including how we split the kernel in different experiments, the suitability metrics and the platforms used.

C.3.1 Experimental Setup

We use the classes specified above to define three experiments — A, B and C —, each providing a different level of optimisation effort for HLS. For FPGA, experiments A, B and C include kernels from classes I, II and III, respectively. For GPUs, we use the most adequate kernel versions available in the classes. For example, it is not suitable to use the kernels of class III on a GPU, since all kernels use just a single compute core, wasting all the SIMD potential of this architecture. For the Rodinia and SHOC kernels, we always use the kernels from class I on GPU. For the remaining kernels, we use the NDRange versions from class II. Although these were optimised for FPGA, they still adopt the correct execution model for GPUs. Chart 20 presents how the classes are related to the

proposed experiments.

Chart 20 – Class mapping for each experiment.

Experiment	FPGA	GPU
A	Class I	Class I
B	Class II	Class II[a] and Class I[b]
C	Class III	Class I

[a] Poisson(1), Poisson(2), Bitonic(1), Bitonic(2) and Bitonic(3) only.
[b] For all remaining kernels in experiment B.
Source: Elaborated by the author.

Some of the applications here considered are split into multiple kernels. These kernels are identified numerically between parentheses (e.g. NW(1)). While in classes I and II the NW application is composed of two kernels, Zohouri (2018) merged it into a single one for the task-optimised version. In this case, we compare the performance of NW from experiment C against the performance of NW(1) and NW(2) together. Since the host logic is minimal between the kernel calls for NW(1) and NW(2), we believe that such comparison is fair.

Our intention is to compare the sole compute capability of each platform, therefore we only measure the execution time of the kernel, ignoring host and data transfer times.

C.3.2 Suitability Metrics

For the evaluation, we use two metrics: execution time t and estimated consumed energy e. We estimate consumed energy by multiplying t and p:

$$e = p \times t \tag{C.1}$$

where p is the platform's Thermal Design Power (TDP).

A straightforward evaluation is to compare t or e from each platform, then assess which is the most suitable platform by considering the smallest value. This, however, does not quantify how much "better" or "worse" one platform is in relation to another for a given kernel. Thus we propose an additional evaluation where the "relative distance" of each metric is considered. Considering two platforms of comparison x and y, and a metric m_i to be compared ($i \in \{x, y\}$, $m \in \{t, e\}$), we define the relative distances d_x and d_y as being:

$$d_x = \frac{m_y}{m_x + m_y} \tag{C.2a}$$

$$d_y = \frac{m_x}{m_x + m_y} \tag{C.2b}$$

$$1 = d_x + d_y \tag{C.2c}$$

If both platforms have similar results for a metric, d_x and d_y will tend to 0.5, i.e. they have similar performance. If $m_x < m_y$ d_x will tend to 1 while d_y will tend to 0. Therefore, d_x and d_y quantitatively express the suitability (in terms of m) of each platform for a given kernel. We will henceforth call these distances as "suitability", and we call the analysis performed using these values as **"relative analysis"**. Conversely, we call the straightforward comparison of simply selecting the smallest value as **"absolute analysis"**.

As an example, consider that a kernel had an execution time of 0.9s on platform x and 0.8s on platfom y. The absolute analysis is the simple case of concluding that y is more suitable than x, since $0.8 < 0.9$. The suitability values are $d_x = 0.8 \times (0.8+0.9)^{-1} = 0.47$ and $d_y = 0.9 \times (0.8+0.9)^{-1} = 0.53$. Since both values are close to 0.5, it is possible to assess that both x and y platforms have near-comparable performance results, even though y is faster than x for this kernel.

C.3.3 Accelerator Platforms

We execute our experiments in two FPGAs and two GPUs, as presented in Chart 21. To reduce verbosity, we will refer to each platform by their aliases as defined in the table.

Chart 21 – Platforms used.

Type	Platform	Chipset	TDP	Fab.	Alias
FPGA	BittWare S5PH-Q	Intel FPGA Stratix V 5sgxa7	25W[a]	28nm	`sv`
FPGA	HARP	Intel FPGA Arria 10	30W[b]	20nm	`a10`
GPU	NVIDIA Quadro	NVIDIA Quadro K620	41W[c]	28nm	`qdr`
GPU	EVGA ACX 2.0	NVIDIA GTX980	165W[d]	28nm	`gtx`

[a] (Nallatech, 2018a)
[b] (Nallatech, 2018b; Intel Corporation, 2018a)
[c] (NVIDIA Corporation, 2018)
[d] (Geeks3D, 2015)
Source: Elaborated by the author.

These two GPUs were selected as being representative of both low and high power platforms. The `gtx` is considered as a high-power, high-performance GPU, whereas `qdr` has lower power.

Table 10 – Compilation and execution success results.

		sv		a10		qdr		gtx	
Exp.	Total	Synth.	Exec.	Synth.	Exec.	Compil.	Exec.	Compil.	Exec.
A	38	34	34	34	20	38	36	38	36
B	14	13	13	14	13	14	13	14	13
C	4	4	4	3	3	3	3	3	3

Source: Research data.

The `a10` FPGA is available as part of the Intel Hardware Accelerator Research Program (HARP) (Intel Corporation, 2018b). For some of the kernels optimised by Zohouri (2018), a version specifically tailored for the Arria 10 FPGA is provided. Therefore we use these versions when targetting the `a10` FPGA.

We could not find the typical TDP for the exact FPGA platforms we used, therefore we used TDP from different platforms that share the same chipset (references in Chart 21). For the GPUs, we used NVIDIA OpenCL SDK 384.81. For the FPGAs, we used Intel FPGA SDK for OpenCL versions "16.1" and "16.0 Pro" for `sv` and `a10`, respectively.

C.4 Results

Not all kernels did synthesise or run on both platforms due to various reasons, such as lack of enough FPGA resources or compiler failures (e.g. compiler segmentation fault)[2]. In the following sections, we analyse the results for each experiment through 2 comparison scenarios: `sv` vs. the GPUs and `a10` vs. the GPUs. Therefore, we only consider the kernels that successfully ran on all platforms for each scenario (e.g. when analysing `sv` vs. the GPUs, we consider the kernels that executed on `sv`, `qdr` and `gtx`). Table 10 presents the number of kernels that successfully executed in each platform.

C.4.1 Experiment A

Considering the 38 kernels from the first experiment, 32 successfully executed on `sv`, `qdr` and `gtx`, and 20 successfully executed on `a10`, `qdr` and `gtx`. Table 11 presents the absolute analysis (as explained in subsection C.3.2) for both proposed scenarios. None of the FPGAs performed better in terms of execution time for any kernel, however the figure slightly changes when analysing the energy consumption. For some kernels, the FPGAs were more energy-efficient than the high-power `gtx` GPU. However, the low-power `qdr` still beats all FPGAs in both metrics. Although the GPUs clearly performed better, the

[2] Please refer to the project repository for more detailed information regarding the failures.

Table 11 – Experiment A: absolute analysis.

Suitability Criterion	Execution time				Energy consumption			
Comparison	sv × qdr	sv × gtx	a10 × qdr	a10 × gtx	sv × qdr	sv × gtx	a10 × qdr	a10 × gtx
# of FPGA-suitable kernels	0	0	0	0	0	6	0	2
# of GPU-suitable kernels	32	32	20	20	32	26	20	18

Source: Research data.

few kernels where FPGA had some advantage may show that the NDRange model is still viable for FPGAs.

Table 12 presents the results for relative analysis using the proposed suitability metric. Kernels that did not execute on each FPGA and both GPUs are filled with "—". Since d_{fpga} is complementary to d_{gpu}, we only present the d_{fpga} values (i.e. the higher the values, the more FPGA-suitable a kernel is). The average suitability shows that both FPGAs were considered less efficient when compared to the low-power `qdr` (0.17 and 0.11 for `sv` and `gtx`) than when compared to the `gtx` (0.27 and 0.19 for `sv` and `a10`, respectively). Figure 56 presents a graphical representation for the comparison, where it can be seen that the FPGA suitability tends to be reduced when compared to the low-power GPU. Further analysis is required using actual energy measurements to draw any conclusions in this end.

Comparing both FPGAs, `sv` presented better results than `a10` for nearly every kernel. However, the `a10` is able to house more complex hardware (e.g. GEMM).

C.4.2 Experiment B

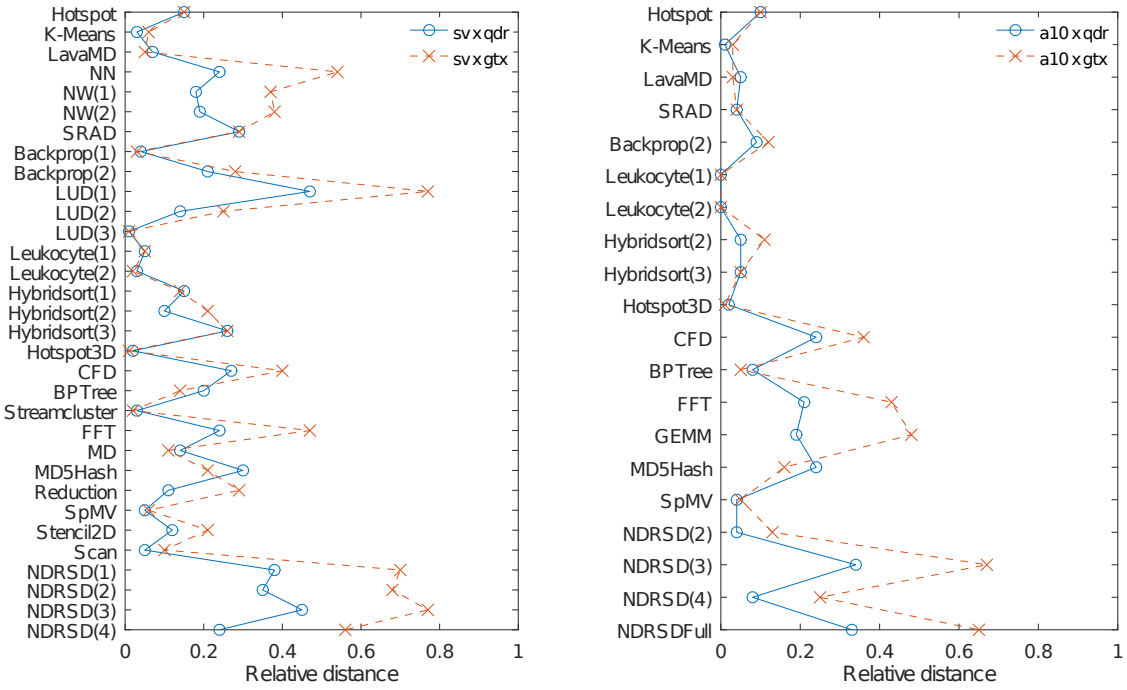
From the 14 FPGA-optimised NDRange kernels, 12 succesfully executed on `sv`, `qdr` and `gtx`, and also 12 successfully executed on `a10`, `qdr` and `gtx`. Table 13 presents the absolute analysis for both proposed scenarios. The Poisson kernels is more efficient on the `sv` FPGA than on the `gtx` GPU. This result is divergent from the original work (WELLER *et al.*, 2017), since in their case the FPGA loses in both performance and energy consumption against a high-power GPU. However, they used CUDA for the GPU version, which may provide better results than using OpenCL on GPU. All the Bitonic kernels performed worse in FPGA, which is also contrary to the original results from Muslim *et al.* (2017). However, we removed optimisations that were incompatible to the vendor we used (Altera). This likely led to performance degradation and further indicates the need of platform and vendor-specific optimisations. The Hotspot kernel from this experiment is actually the most optimised version from Zohouri (2018) and therefore it presents better

Table 12 – Experiment A: relative analysis.

Kernel	`sv` × `qdr`	`sv` × `gtx`	`a10` × `qdr`	`a10` × `gtx`
Hotspot	0.15	0.15	0.10	0.10
K-Means	0.03	0.06	0.01	0.03
LavaMD	0.07	0.05	0.05	0.03
NN	0.24	0.54	—	—
NW(1)	0.18	0.37	—	—
NW(2)	0.19	0.38	—	—
Pathfinder	—	—	—	—
SRAD	0.29	0.29	0.04	0.04
Backprop(1)	0.04	0.03	—	—
Backprop(2)	0.21	0.28	0.09	0.12
LUD(1)	0.47	0.77	—	—
LUD(2)	0.14	0.25	—	—
LUD(3)	0.01	0.01	—	—
Leukocyte(1)	0.05	0.05	0.00	0.00
Leukocyte(2)	0.03	0.02	0.00	0.00
Hybridsort(1)	0.15	0.14	—	—
Hybridsort(2)	0.10	0.21	0.05	0.11
Hybridsort(3)	0.26	0.26	0.05	0.05
Hotspot3D	0.02	0.01	0.02	0.01
CFD	0.27	0.40	0.24	0.36
BPTree	0.20	0.14	0.08	0.05
Particlefilter(1)	—	—	—	—
Particlefilter(2)	—	—	—	—
Streamcluster	0.03	0.02	—	—
BFS	—	—	—	—
FFT	0.24	0.47	0.21	0.43
GEMM	—	—	0.19	0.48
MD	0.14	0.11	—	—
MD5Hash	0.30	0.21	0.24	0.16
Reduction	0.11	0.29	—	—
SpMV	0.05	0.06	0.04	0.05
Stencil2D	0.12	0.21	—	—
Scan	0.05	0.10	—	—
NDRSD(1)	0.38	0.70	—	—
NDRSD(2)	0.35	0.68	0.04	0.13
NDRSD(3)	0.45	0.77	0.34	0.67
NDRSD(4)	0.24	0.56	0.08	0.25
NDRSDFull	—	—	0.33	0.65
Average	0.17	0.27	0.11	0.19

Source: Research data.

Figure 56 – Relative analysis for experiment A comparing each FPGA against both GPUs.



(a) `sv` × *GPU* (b) `a10` × *GPU*

Source: Research data.

Table 13 – Experiment B: absolute analysis.

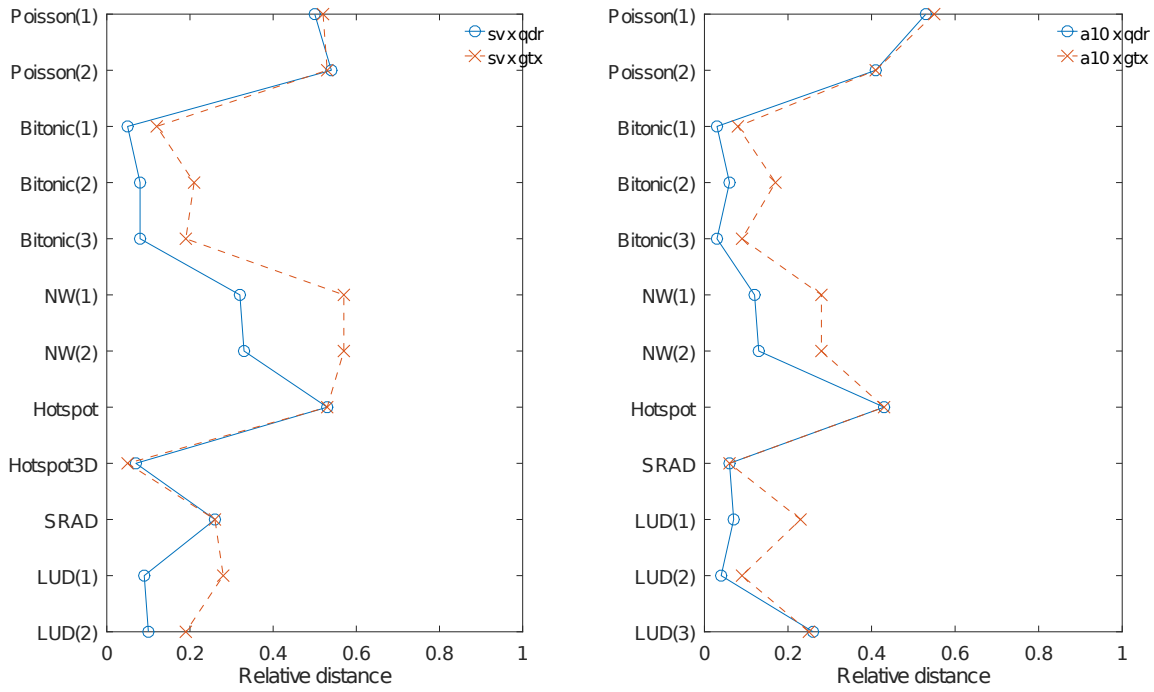
Suitability Criterion	Execution time				Energy consumption			
Comparison	sv × qdr	sv × gtx	a10 × qdr	a10 × gtx	sv × qdr	sv × gtx	a10 × qdr	a10 × gtx
# of FPGA-suitable kernels	0	0	0	0	2	5	1	1
# of GPU-suitable kernels	12	12	12	12	10	7	11	11

Source: Research data.

energy efficiency than the GPUs for the `sv` FPGA. Similarly to the previous experiment, the `a10` FPGA performed worse than the `sv` for nearly every kernel.

Table 14 presents the results for relative analysis. The outcome is quite similar from the first experiment: the FPGAs are in a slightly better position against the `gtx` than the `qdr` GPU. However, the FPGAs still lose in performance for every kernel against GPUs. Nonetheless, the average suitability from both FPGAs increased when compared to the previous experiments, which corroborates with the need of FPGA-specific optimisations. Figure 57 presents the graphical representation of the relative analysis.

Figure 57 – Relative analysis for experiment B comparing each FPGA against both GPUs.



(a) `sv` × *GPU* (b) `a10` × *GPU*

Source: Research data.

C.4.3 Experiment C

Finally, the 4 task-optimised kernels were executed on FPGA. A total of 3 kernels succesfully executed on `sv`, `qdr` and `gtx`, and only 2 successfully executed on `a10`, `qdr` and `gtx`. Table 15 presents the absolute analysis for both proposed scenarios. The NW kernel did not fit on the `a10`, requiring further optimisation study. For the first time in the experiments, one kernel had better performance on the `sv` FPGA. As mentioned in the previous experiment, the NDRange version of Hotspot is more optimised than the task version even for FPGA, therefore it presents worse performance in this experiment than the previous one. The optimisation applied to Hotspot3D greatly improved its standpoint, however still not enough to beat the GPUs.

Table 16 presents the results for relative analysis. The `sv` presents an improved average suitability compared to the other experiments, whereas not much changed for the `a10`. By analysing the compilation reports, we found that the optimisations applied in these kernels are enough for generating efficient pipelines in both FPGAs, but `a10` designs had slower operating frequencies.

Table 14 – Experiment B: relative analysis.

Kernel	sv × qdr	sv × gtx	a10 × qdr	a10 × gtx
Poisson(1)	0.50	0.52	0.53	0.55
Poisson(2)	0.54	0.53	0.41	0.41
Bitonic(1)	0.05	0.12	0.03	0.08
Bitonic(2)	0.08	0.21	0.06	0.17
Bitonic(3)	0.08	0.19	0.03	0.09
NW(1)	0.32	0.57	0.12	0.28
NW(2)	0.33	0.57	0.13	0.28
Hotspot	0.53	0.53	0.43	0.43
Hotspot3D	0.07	0.05	—	—
Pathfinder	—	—	—	—
SRAD	0.26	0.26	0.06	0.06
LUD(1)	0.09	0.28	0.07	0.23
LUD(2)	0.10	0.19	0.04	0.09
LUD(3)	—	—	0.26	0.25
Average	0.25	0.34	0.18	0.24

Source: Research data.

Table 15 – Experiment C: absolute analysis.

Suitability criterion	Execution time				Energy consumption			
Comparison	sv × qdr	sv × gtx	a10 × qdr	a10 × gtx	sv × qdr	sv × gtx	a10 × qdr	a10 × gtx
# of FPGA-suitable kernels	1	1	0	0	1	1	0	0
# of GPU-suitable kernels	2	2	2	2	2	2	2	2

Source: Research data.

Table 16 – Experiment C: relative analysis.

Kernel	sv × qdr	sv × gtx	a10 × qdr	a10 × gtx
NW	0.83	0.93	—	—
Hotspot	0.20	0.19	0.14	0.14
Hotspot3D	0.32	0.25	0.26	0.20
Pathfinder	—	—	—	—
Average	0.45	0.46	0.20	0.17

Source: Research data.

Table 17 – Average execution time ratios all vs. all[ab].

	sv	a10	qdr	gtx	sv-B	a10-B	sv-C	a10-C
sv		1.82	—	—	—	—	—	—
a10	—		—	—	—	—	—	—
qdr	20.66	116.36		—	3.80	4.84	—	5.43
gtx	82.44	575.16	2.87		9.56	14.35	1.47	27.41
sv-B	60.54	6.83	—	—		1.37	—	—
a10-B	107.55	4.32	—	—	—		—	2.62
sv-C	135.60	13.79	1.12	—	4.23	9.60		1.11
a10-C	179.85	11.90	—	—	2.12	—	—	

[a] The number at row i and column j means how faster or more efficient i is against j.
This matrix is symmetrical, thus the values where j is better than i are ommitted.

[b] The naming style `x-Z` is used in the rows and columns to refer to the kernels from experiment Z on the platform x (e.g. `a10-B` are the kernels from experiment B when executed on the `a10`).

Source: Research data.

Table 18 – Average energy consumption ratios all vs. all.

	sv	a10	qdr	gtx	sv-B	a10-B	sv-C	a10-C
sv		2.18	—	—	—	—	—	—
a10	—		—	—	—	—	—	—
qdr	12.60	85.14		1.40	2.32	3.54	—	3.97
gtx	12.49	104.57	—		1.45	2.61	—	4.98
sv-B	60.54	8.20	—	—		1.65	—	—
a10-B	89.62	4.32	—	—	—		—	2.62
sv-C	135.60	16.54	1.83	4.50	4.23	11.52		1.33
a10-C	149.88	11.90	—	—	1.77	—	—	

Source: Research data.

C.4.4 Final Comparison

Table 17 and Table 18 present the average execution time and power ratios between the accelerators. For each possible pair of accelerators (x, y)[3], we selected the kernels that executed on both, calculated the execution time and energy consumption ratio for every kernel and extracted the average.

For both execution time and energy consumption, the `sv` only has a ratio greater than 1 against GPUs when considering the aggressively-optimised experiment C (`sv-C`). In all other cases, the `sv` only beats the other FPGA `a10`.

[3] Where $x, y \in \{\texttt{sv}, \texttt{a10}, \texttt{qdr}, \texttt{gtx}\}, x \neq y$.

In every experiment, the older `sv` did perform better for almost every kernel when compared to `a10`. We noticed that in average the operating frequency of kernels using `a10` was 13%, 14% and 1% smaller than `sv` for experiments A, B and C, respectively. We benchmarked sequential reads and writes to global memory from within the kernel programmable space[4], which peaked at 17.48 GB/s for `sv` and 15.08 GB/s for `a10`. Even though `a10` uses newer DDR4 memory than `sv` DDR3, the reduced frequency and memory bandwidth of `a10` have a negative impact on memory-bound kernels. However, `a10` was able to house more resource-hungry kernels which did not fit in `sv`.

Although we do not consider host-device communication latency in our study as we are comparing compute power, it is a significant overhead for many applications. The `a10` FPGA used in this work is part of the Intel HARP featuring a Xeon CPU tightly-coupled to an Arria 10 FPGA, host-device data exchange latency is greatly reduced as no communication via peripheral bus is needed, differently from our `sv` FPGA that uses PCI-Express.

C.4.5 Discussion

All findings pointed to a similar trend from the related work: hardware-specific optimisation are crucial to improve the FPGA's energy efficiency when using HLS. Even more effort is required if better performance is also desirable. In the case of OpenCL, GPUs have been longer supported than FPGAs. Most existing kernels are NDRange, since they originally target GPUs. Which execution model suits best for FPGA is application-dependent, depending on memory access, computation pattern and how well the HLS compiler can extract hardware parallelism from the software code. As pointed by Zohouri (2018), some Rodinia kernels would not benefit from great speedups by using the NDRange kernel, therefore a complete rewrite using the task model was performed by the authors.

Although the goal of HLS compilers is to minimise the hardware design burden when programming to FPGAs, the type of aggressive optimisations applied in the kernels presented still requires a great amount of hardware design knowledge. Additionally, in several kernels FPGA performance is bottlenecked by the global memory, usually DDR3. Faster memories is highly desirable, since modern GPUs use high-speed memories with wide buses. This is being addressed by the industry through the inclusion of High-Bandwidth Memory (HBM) modules on FPGAs, however they are still limited to the top grade families (Intel FPGA Stratix 10, Xilinx Virtex UltraScale+ HBM);

Even with these drawbacks, we still believe that FPGAs have the potential to increase its role as an application accelerator in the ever-increasing dark silicon era. FPGAs excel in deep customised pipelines, and highly-optimised FPGA circuits (most developed

[4] Sequential 64 bytes read and writes using transfer sizes of 1, 8, 16 and 64 MB.

in RTL languages) are still very competitive in both performance and energy efficiency when compared to other traditional accelerators.

C.5 Final Remarks

This chapter presented a wide comparison study performed between FPGA and GPU as accelerators using OpenCL. A set of 56 OpenCL kernels was gathered from various sources in the literature, with varying optimisation efforts towards FPGA. Results show that aggressive optimisations are essential for performance and energy improvements in FPGAs when compared to GPUs. The OpenCL HLS model for FPGAs still requires several improvements in order to bring the FPGA as a fair competitor in the high-performance computing scenario. Skilled hardware expertise was required to efficiently model the kernel codes that we gathered.

Our former intention was to provide a data mining model capable of evaluating the suitability of an existing OpenCL kernel when targetting FPGAs. This model would consider that the kernels would be optimised using an automated framework. However, the amount and complexity of the optimisations herein presented have shown that the automated framework would require significant and possibly unviable implementation effort.

Most OpenCL kernels available were designed with a platform in mind, such as GPU. These codes are often unsuitable for HLS compilation, requiring significant optimisation effort as previously shown. Therefore we believe that the sequential algorithm of the problem is a better input for HLS compilation, since it provides a clearer structure of potential deep pipeline possibilities. To this reason, we decided to step back from OpenCL and use C/C++ as the input language.

The results here presented also show a trend that FPGAs appear to be more competitive against high-power GPUs than low-power. For example, the FPGAs used were in a better performance or energy standpoint when compared against the high-power `gtx` than when compared to the low-power `qdr`. Most related work only compares against high-power GPUs, which could indicate a general biased trend. However, further analysis is required in order to draw any conclusions. For example, energy should be measured instead of using the TDPs as approximation. Nonetheless, the results here presented were already enough to model the next steps towards HLS optimisation, and we decided not to proceed on energy measurement up to this point.

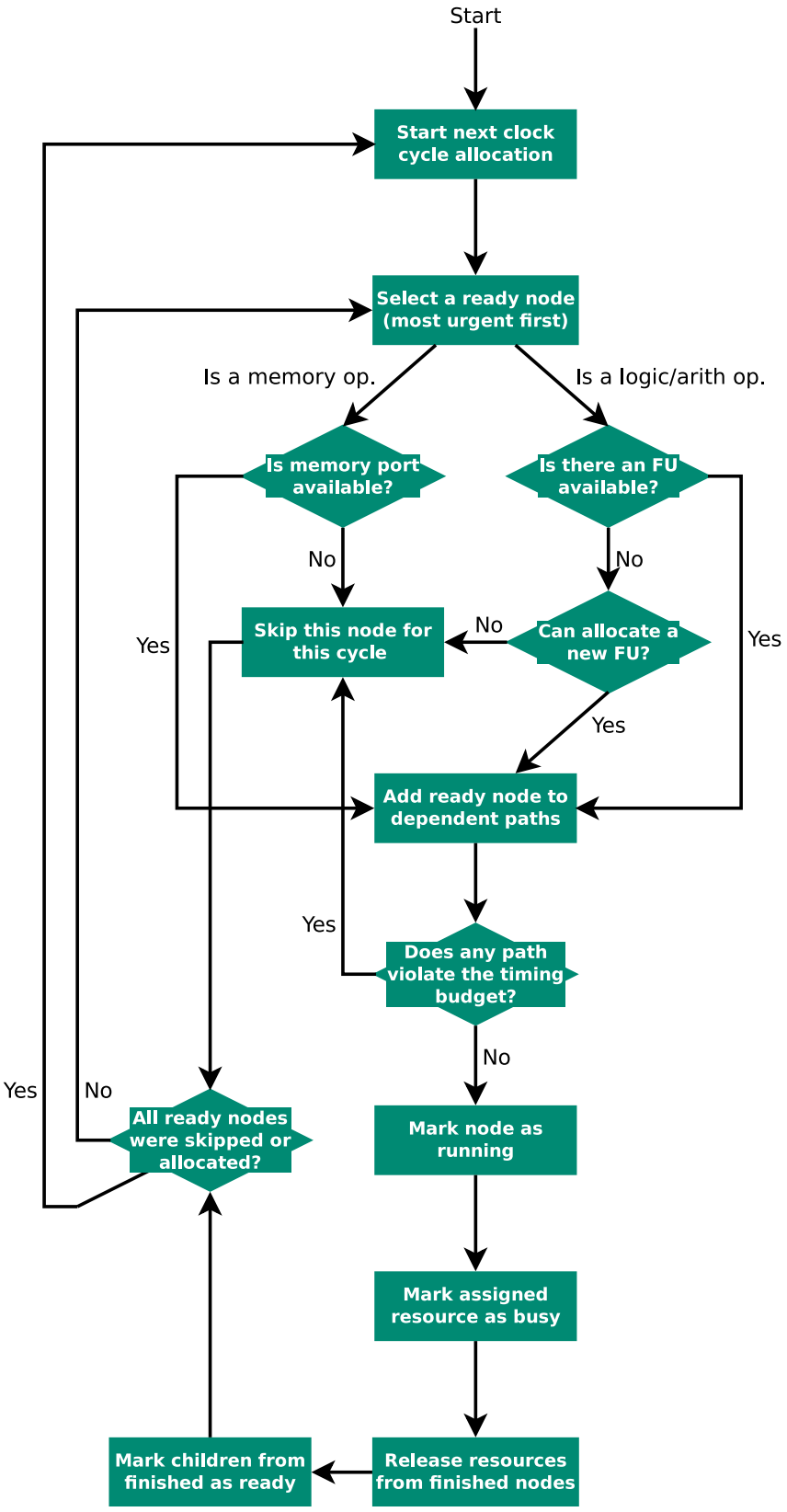
APPENDIX

D

LINA DSE: RESOURCE AND TIMING-CONSTRAINED SCHEDULER FLOWCHART

Figure 58 presents the resource and timing-constrained scheduling flowchart (next page due to size).

Figure 58 – Flowchart of the resource and timing-constrained scheduling performed by Lina. The urgency of a ready node is defined by the ALAP scheduling results (lower values are more urgent).



Source: Perina *et al.* (2021).

APPENDIX

E

OPTIMISED FPGA-GPU COMPARATIVE ANALYSIS

In this appendix, we perform a FPGA vs. GPU comparison similar to the one presented in Appendix C, however now on a different scenario and approach. Both platforms will execute the same application, but using appropriate optimisations and high-level languages for each. On the FPGA side, we use Lina to perform DSE on the C/C++ version of these applications. On the GPU side, we execute the CUDA version. At last, we compare considering both design execution time and consumed energy.

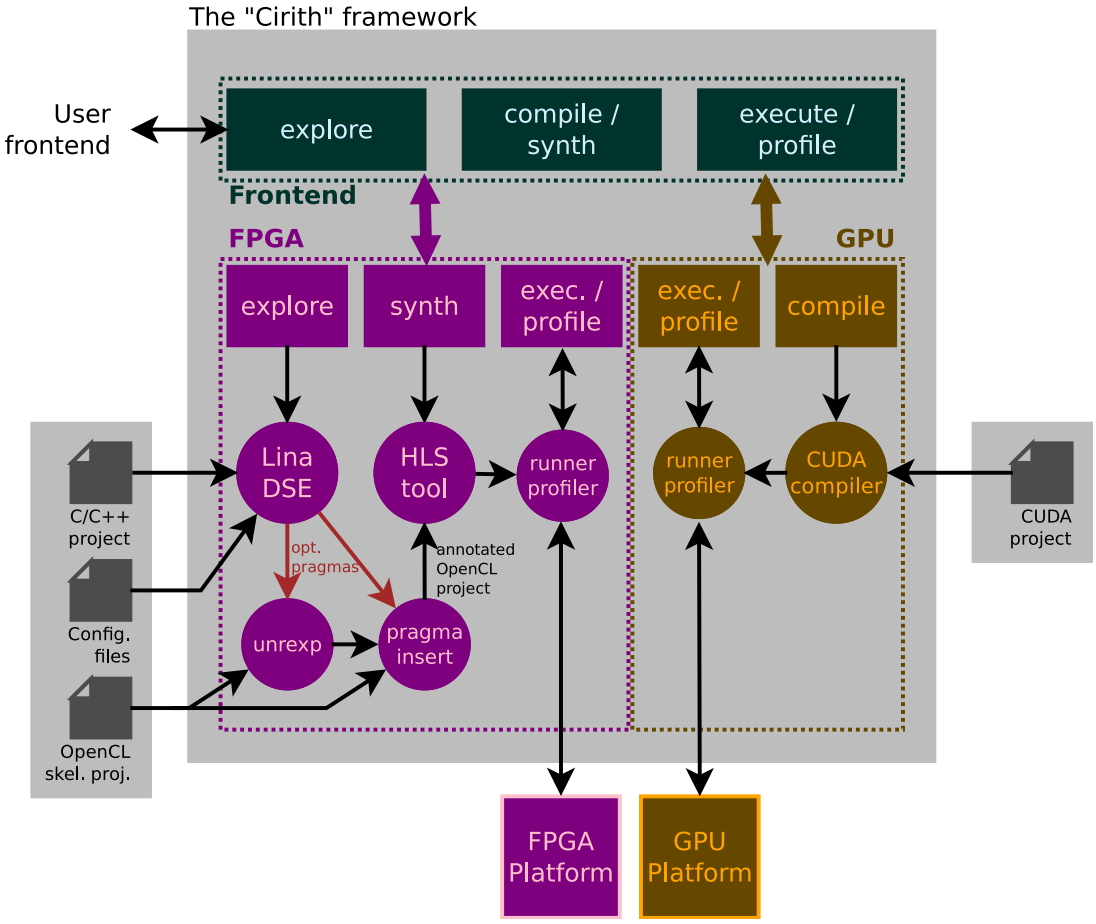
In section E.1 we present the experimental framework used for comparison, including tools, platforms, and the benchmark used. Then, section E.2 presents the comparison results and related discussion. Finally, section E.3 closes the appendix.

E.1 Experimental Framework

Figure 59 presents an overview of the framework used in this chapter. It is a single frontend used to coordinate both accelerators. Each application tested using this framework is composed of two projects, one for FPGA and another for GPU. On the FPGA side, a C/C++ baseline project is used by Lina for exploration. Then, a skeleton OpenCL project is annotated with the optimal configuration estimated by Lina. This skeleton project is based on the C/C++ baseline project, and the computation kernel is wrapped as an OpenCL kernel. The GPU side is a CUDA variant of the baseline implementation. The host code and CUDA kernels are manually coded and optimised.

In order to avoid the "read-after-write" code pattern during unroll, we adapted the explicit unroll logic used for the CNN kernels and created a tool named `unrexp`. However, the OpenCL skeleton project must be specifically adapted for this tool through annotation and code rewrite. If the skeleton project is not supplied in this format, the `unrexp` tool is

Figure 59 – Overview of the experimental framework.



Source: Elaborated by the author.

simply bypassed and normal HLS unroll directives are used (i.e. `unrviv`).

In order to provide more accuracy than estimating the consumed energy using the TDP value as in Appendix C, we now perform actual power measurement for the energy consumption analysis. In the following sections, we present the platforms used for the experiment and the components used for power profiling.

E.1.1 FPGA Platform

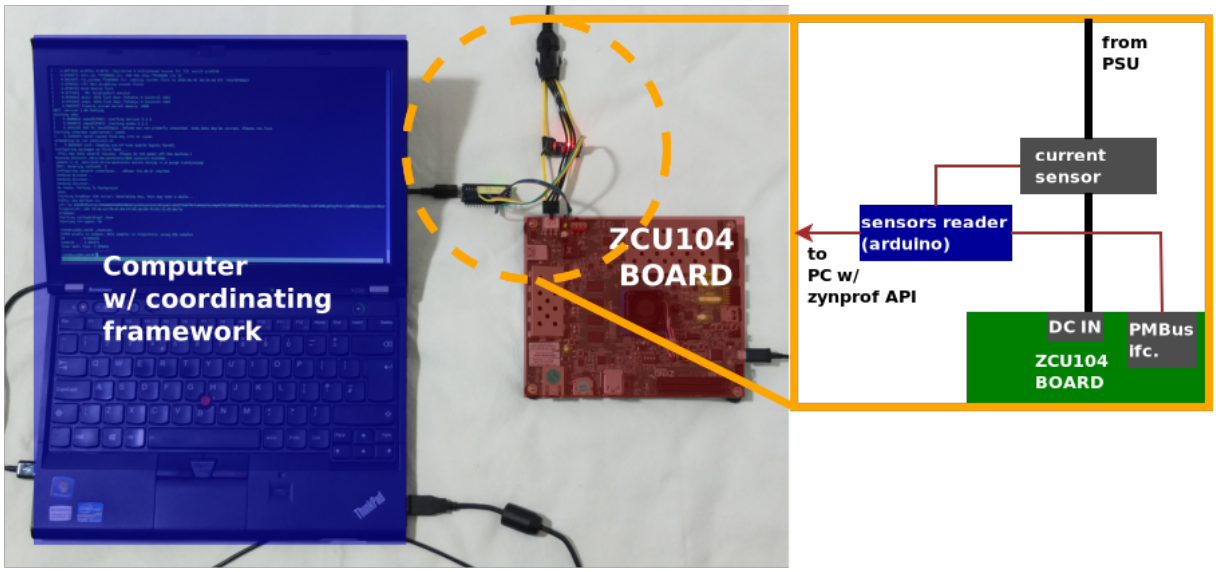
We use the same Xilinx Zynq UltraScale+ ZCU104 board from previous chapters, since our framework already supports this platform. Our framework is responsible for preparing the project as a bootable SD card system, interact with the Linux-based system on the board, execute, retrieve the results after execution, and collect the power profiling metrics.

The ZCU104 board has 8 power regulators, however only two have accessible PM-Bus[1] interfaces (Xilinx, Inc., 2018). With a special external module, it is possible to read

[1] Power Management Bus (PMBus) is a standardised communication protocol used to interact

the PMBus interfaces and collect energy information, however the other 6 regulators remain unmetered. In order to acquire the complete energy consumption of the board, we attached a current sensor between the board and its power supply. Our framework is responsible for coordinating when to enable/disable power sensing, and to synchronise the profiled values with the kernel execution on board. Figure 60 presents the FPGA platform, including the additional modules used for power sensing.

Figure 60 – FPGA experimental setup. The power sensing system is highlighted. An Arduino module reads the PSU current sensor and communicates with the regulators using the PMBus interface. The raw information is collected and sent to the host machine, which calculates consumed energy using a tool named `zynprof`.



Source: Elaborated by the author.

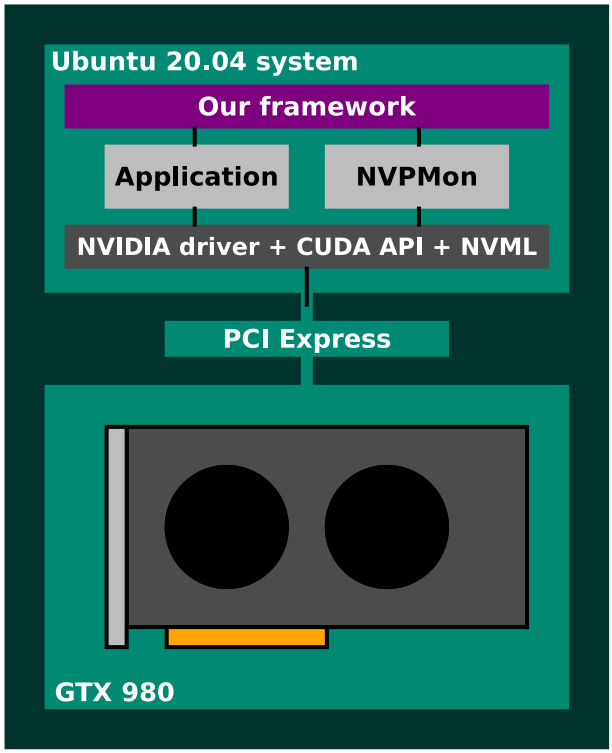
The current sensor between the board and the power supply does not measure voltage. Since knowing the voltage is required for calculating the energy consumption, we acquire the input voltage of all accessible regulators at runtime. Then, we average this value and use it as total input voltage. Alternatively, a constant value of 12V could be used (i.e. the power supply's rated output voltage), however the use of a constant value disregards the voltage fluctuations over the execution time.

E.1.2 GPU Platform

On the GPU side, we use the GeForce GTX980 board connected to a host system running Ubuntu 20.04 on an Intel Xeon E5-1603 CPU. Power sensing is performed by using NVIDIA's Management Library (NVML) API. Similarly to the FPGA, our framework is responsible for coordinating the power sensing and synchronise the values with the kernel execution. Figure 61 presents the GPU system.

with power devices.

Figure 61 – GPU experimental setup. Our framework coordinates the application and extracts the power measurements from NVIDIA's NVML using our in-house tool (`NVPMon`).



Source: Elaborated by the author.

E.1.3 Kernel Set

For this experiment, it is desirable to have proper C/C++ and CUDA variants for each test application. The Parboil benchmark is composed of several applications with multiple implementations, such as: a baseline C/C++ implementation; CPU-optimised versions using OpenMP; and GPU variants on CUDA or OpenCL. We use the baseline C/C++ implementations as input for Lina and the HLS process, whereas we use the CUDA versions for GPU (if more than one CUDA version is available, we use the most optimised). Five kernels were selected that are compatible with Lina, considering the limitations presented in section 3.5. Chart 22 describes the kernels included in our experiments and their dimensions[2].

Chart 23 presents the design space of each. Since `unrexp` currently only supports unrolling the innermost loop, the kernel versions adapted for the tool may have different knobs.

We map all arrays that have read or write accesses (but not both) to the off-chip memory, whereas we leave arrays with both read and write accesses in the on-chip memory. This reduces the chances of `read-after-write` code patterns from occurring. The arrays accessed on-chip by the computation kernel must first be transferred from off-chip to

[2] Kernel dimensions are constant values used to define arrays and loop sizes.

Chart 22 – Last experiment kernel set.

Kernel name	Description	Dimensions
`histo`	Histogramming operation	**Image size:** 996×1040 **Histogram size:** 256×4096 **Number of iterations:** 200
`lbm`	Lattice-Boltzman Method simulation	`SIZE_X`: 120 `SIZE_Y`: 120 `SIZE_Z`: 30
`mri-q`	MRI non-cartesian Q matrix calculation	`K`: 256 `X`: 32768
`sad`	Sum of Absolute Differences	**Macroblock size:** 11×9 **Search range:** 16 **Max. search positions:** 1089
`sgemm`	Single precision general matrix multiply	`M`: 256 `N`: 304 `K`: 192

Source: Elaborated by the author.

on-chip memory, and then retrieved after execution. Our framework automatically adds loops before and after the computation kernel in order to perform such data movement.

For each kernel, we provide two FPGA versions. The first version uses the code based on the original baseline code, whereas the second version is modified to support the `unrexp` manipulation. For each of these FPGA versions, we provide four Lina explorations by toggling data packing and the `conservative`/`permissive` policies[3]. We leave banking enabled in all cases.

We perform two experiments when running the kernels on GPU, named "cold run" and "hot run". When the applications are executed, the GPU switches from a low-profile to a higher energy profile in order to provide more compute capability. This in turn increases the energy consumption. We get the "cold" values by executing the kernels with a cooldown interval, and the "hot" values by executing the kernels repeatedly with no cooldown intervals.

For each application, we added configuration files needed by our framework to characterise the design spaces and to configure other Lina settings. The baseline C/C++ projects were also adapted for HLS compilation (e.g. wrapping the computation kernel as an OpenCL kernel). The following subsection presents how each project is structured and the modifications performed to each kernel.

[3] Please note that even if Lina's data packing exploration is disabled, Vivado might still automatically vectorise the kernels.

Chart 23 – DSE knobs for the Parboil kernels.

Kernel	Knobs
`histo`	**Frequencies (MHz):** 75, 100, 150, 200 **Unroll factors (single loop level):** none, 2, 4, 8, 16, 20, 40, 80, 160, 320 **Pipelining (single loop level):** allowed **Partitioning (array `hist`):** block (4, 8, 16), cyclic (4, 8, 16, 32)
`histo` (`unrexp`)	Same as normal version
`lbm`	**Frequencies (MHz):** 75, 100, 150, 200 **Unroll factors (single loop level):** none, 3, 9, 18, 27, 36 **Pipelining (single loop level):** allowed
`lbm` (`unrexp`)	**Frequencies (MHz):** 75, 100, 150, 200 **Unroll factors (single loop level):** none, 3 **Pipelining (single loop level):** allowed
`mri_q`	**Frequencies (MHz):** 75, 100, 150, 200 **Unroll factors (loop level 1):** none, 2, 4, 8 **Unroll factors (innermost):** none, 4, 8, 32 **Pipelining (innermost):** allowed **Partitioning (array `Qr`):** block (2, 8), cyclic (2, 8) **Partitioning (array `Qi`):** block (2, 8), cyclic (2, 8)
`mri_q` (`unrexp`)	**Frequencies (MHz):** 75, 100, 150, 200 **Unroll factors (innermost):** none, 4, 8, 32 **Pipelining (innermost):** allowed **Partitioning (array `Qr`):** block (2, 8), cyclic (2, 8) **Partitioning (array `Qi`):** block (2, 8), cyclic (2, 8)
`sad`	**Frequencies (MHz):** 75, 100, 150, 200 **Unroll factors (loop level 7):** none, 2, 4 **Pipelining (loop level 7):** allowed **Unroll factors (innermost):** none, 2, 4 **Pipelining (innermost):** allowed
`sad` (`unrexp`)	**Frequencies (MHz):** 75, 100, 150, 200 **Pipelining (loop level 7):** allowed **Unroll factors (innermost):** none, 2, 4 **Pipelining (innermost):** allowed
`sgemm`	**Frequencies (MHz):** 75, 100, 150, 200 **Unroll factors (loop level 2):** none, 2 **Unroll factors (innermost):** none, 4, 8, 16, 32, 64 **Pipelining (innermost):** allowed **Partitioning (array `C`):** block (2, 8), cyclic (2, 8)
`sgemm` (`unrexp`)	**Frequencies (MHz):** 75, 100, 150, 200 **Unroll factors (innermost):** none, 4, 8, 16, 32, 64 **Pipelining (innermost):** allowed **Partitioning (array `C`):** block (2, 8), cyclic (2, 8)

Source: Elaborated by the author.

E.1.3.1 Project Structuring and Modifications Applied

Based on the multiple variants provided by Parboil for each kernel, the following projects were created and included in our framework:

- **Lina project:** this project is used by Lina for the design space exploration. Minimal modifications were performed, for example:
  - All dynamic loops were converted to have static bounds. Lina partially supports variable loop bounds (e.g. when they are easily identifiable from input data sizes). However, due to Lina's cache system not currently supporting dynamic bound inferring, we maintain all static;
  - Some applications have the kernel code inlined in the host code. We split these projects to always have a host and a kernel code. This is done simply by wrapping the computation loops in a C function;
  - Makefiles were adapted to properly communicate with our framework when compiling, running, profiling, etc.;
  - Configuration files were added to guide Lina's DSE. These files include design space knobs, which objectives should Lina optimise (only design execution time in this case), etc.;
- **Vivado OpenCL template:** in this project, the C kernel function from the Lina project is converted to an OpenCL kernel and annotations are added. Source code 7 presents the template histogram kernel (annotation tags are formatted as `<...>`). The following modifications were performed:
  - The C function header is swapped by an OpenCL kernel header;
  - The beginning and end of kernel function body are annotated with simple tags (e.g. `<HEADER>`). Our framework inserts data transfer loops in these tags when applicable;
  - Each loop level is annotated with a tag identifying its depth. These are replaced by unroll and pipeline directives;
  - All read/write accesses to kernel arguments are replaced by tags containing the array name. Our framework later replaces them with the proper memory space that the computation loops should access (e.g. direct access to off-chip memory or to an on-chip buffer);
  - All loop bounds were also made static. This is not related to Lina's limitation, but to the fact that we were not able to extract Vivado HLS's latency reports without static bounds. Although Vivado supplies directives that can be used to hint the compiler about expected loop bounds, they had no effect;

  - All 2D or 3D arrays were flattened;
  - Lina does not support array of structs. Instead, we split each struct element to a separate array;
  - Makefiles and Vivado HLS scripts were created to coordinate the synthesis process. They were also implemented so that our framework can properly interact with them;
  - OpenCL API calls were added to the host code to manage the off-chip buffers, transfer data to the FPGA, execute the kernel and retrieve the results;

- **Vivado OpenCL template for unrexp:** this project is based on the previous OpenCL one, however the kernel is manipulated so that all reads are placed before writes. Auxiliary variables must be created. Additional annotations are also needed, so that our `unrexp` tool is able to properly replicate the code when unrolling, while maintaining all reads before writes;

- **CUDA project:** Used for the GPU side, the CUDA variants of each Parboil were added with little or no modification on the code itself. Makefiles were adapted so that our framework can properly execute and profile.

Source code 7 – OpenCL kernel template for `histo`

```
1: #define IMG_W 996
2: #define IMG_H 1040
3: #define HISTO_W 256
4: #define HISTO_H 4096
5: #define NUM_ITER 200
6:
7: __attribute__((reqd_work_group_size(1,1,1)))
8: __kernel void histo(
9:   __global uint * restrict img,
10:  __global uchar * restrict hist
11: ) {
12: <HEADER>
13:
14:   for(uint iter = 0; iter < NUM_ITERS; iter++) {
15:     <LOOP_0_1> for(uint i = 0; i < (IMG_W * IMG_H); i++) {
16:       const uint value = <ARR_img>[i];
17:       if(<ARR_hist>[value] < 255)
18:         ++<ARR_hist>[value];
19:     }
20:   }
21:
22: <FOOTER>
23: }
24:
```

Chart 24 presents the additional modifications performed in each kernel specifically. Note that modifications related to data sets are also reflected on the GPU kernels.

E.2 Results

Figure 62 presents the average measured execution times and energy consumptions for each kernel in both platforms. For each, we present all FPGA variants (without/with `unrexp`, toggling memory policies and data packing) and the "cold"-"hot" runs on GPU. We use the same nomenclature as in Chart 10 to identify each exploration. The average value considers five executions for each kernel.

The `histo` kernel failed to synthesise when data packing was enabled. Lina DSE suggested the highest frequency in this case, which caused Vivado to fail during late timing analysis. The `mri_q` kernel failed to synthesise when data packing and `unrexp`

Chart 24 – Modifications performed in each Parboil kernel.

Kernel	Modifications
`histo`	- Number of iterations reduced to 200 - The main iteration loop (former present on host code) was transferred to the OpenCL kernel in order to reduce kernel calls
`lbm`	- `SIZE_Z` reduced from 150 to 30 - Number of iterations reduced to 1
`mri_q`	- Number of iterations reduced to 256 - This kernel uses the trigonometric `sin()` and `cos()` functions. These are currently not supported by Lina. Therefore, we replace these functions by on-chip lookup tables with pre-calculated `sin`/`cos` values - Invariant reads from the innermost loop were transferred to upper loop levels - The error tolerance accepted by the output validation tool was increased from 0.01% to 0.1%. This difference is due to the use of look-up tables
`sad`	- Minor changes when converting from unsigned to signed variables or vice-versa when the words have different sizes. C and OpenCL have slightly different handlings for these cases
`sgemm`	- Statically enforcing $lda = m$, $ldb = n$ and $ldc = m$ - Changed `alpha` to 1.2 and `beta` to 1.3

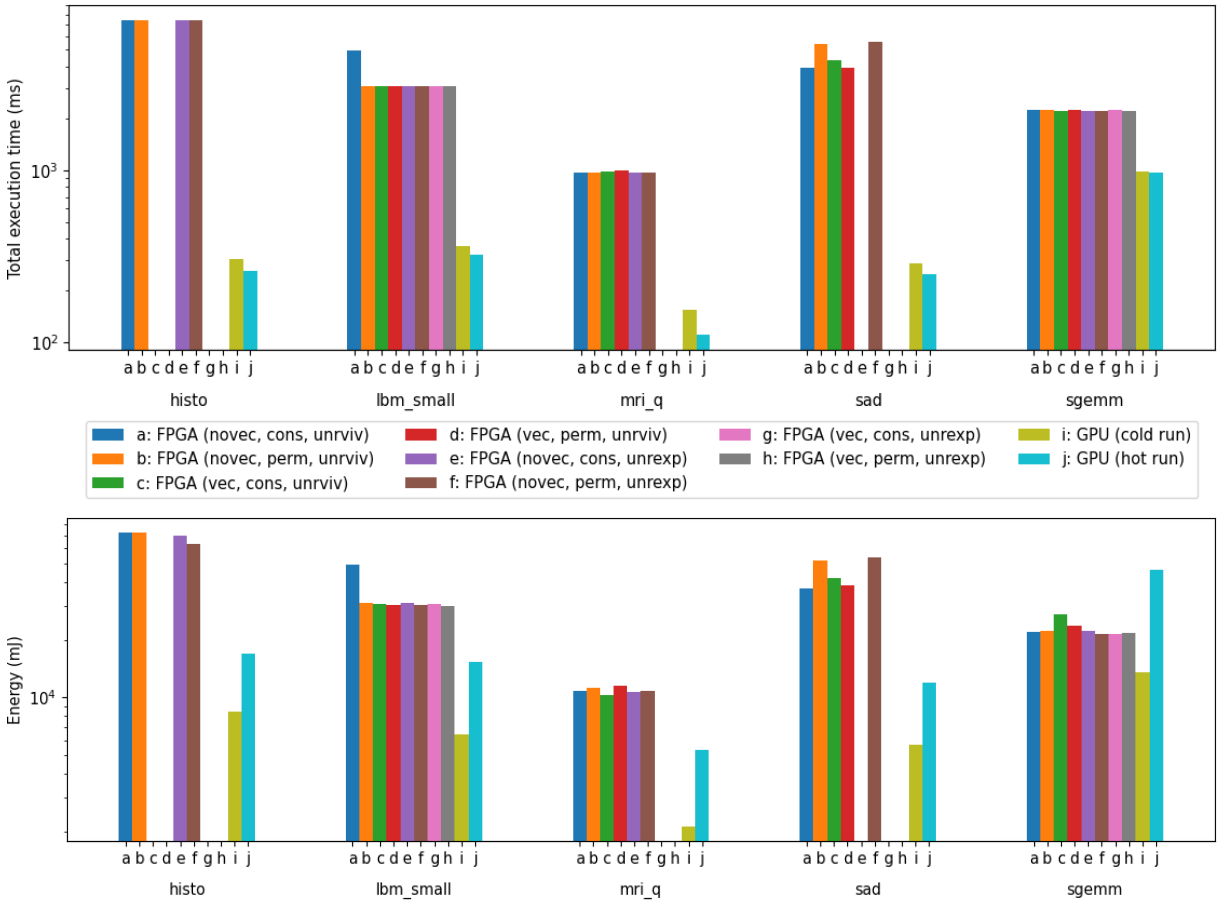
Source: Elaborated by the author.

were enabled. The optimisations suggested by Lina, coupled with the explicit unroll of `unrexp` resulted in an overly complex design that Vivado could not synthesise.

The `unrexp` tool had little or even negative effect on the kernels. Considering the `sad` kernel for example, enabling `unrexp` degraded overall performance. Apart from the variant explored without data packing and using the permissive policy, the execution crashed for all other cases. The version of Vivado/SDSoC used in this thesis has a known bug on which the embedded Linux may kernel panic during long-running kernels[4]. In other words, enabling `unrexp` for the `sad` kernel degraded overall performance up to a level that reaches the crashing point of the embedded Linux system. This also happens with all baseline versions of this same kernel.

[4] See <https://support.xilinx.com/s/question/0D52E00006hpkjESAQ/out-of-memory-when-running-opencl-application-on-zcu102?language=en_US>

Figure 62 – Total execution time (top) and consumed energy (bottom) of each kernel on both platforms. Each is represented by 8 Lina explorations (leftmost bars, from “a” to “h”) and 2 GPU execution scenarios (rightmost bars, “i“ and “j “).



Source: Research data.

The GPU performed better in every kernel considering execution time, and nearly in every kernel when considering energy consumption. The sole exception is kernel `sgemm` that had a better energy efficiency on FPGA, but only when compared against the GPU hot run.

Table 19 and Table 20 present the speedups and energy efficiency gains achieved by Lina solutions compared to the baseline versions[5]. For all kernels tested, the energy consumption ratio between the baseline and Lina solutions followed similar trends, differing at most by 0.7 point between the speedup and energy efficiency gain.

Table 19 – Overall speedups achieved by our optimised kernels compared to the baseline versions.

	auto				expl			
	novec		vec		novec		vec	
Kernel	cons	perm	cons	perm	cons	perm	cons	perm
histo	1.1×	1.1×	—	—	1.1×	1.1×	—	—
lbm_small	0.9×	1.4×	1.4×	1.4×	1.4×	1.4×	1.4×	1.4×
mri_q	2.3×	2.4×	2.3×	2.3×	2.3×	2.4×	—	—
sad[†]	> 3.4×	> 2.5×	> 3.17×	> 3.4×	—	> 2.5×	—	—
sgemm	3.2×	3.2×	3.2×	3.2×	3.2×	3.2×	3.2×	3.2×

[†] The baseline version of `sad` crashes during execution as previously explained. However, we noted that the execution works properly until the crash. We therefore consider that the execution time of the baseline versions is of at least x seconds, where x is the time elapsed until the crash. We then calculate the minimum speedup value that our optimised version achieves when considering the lower bound baseline execution time x.

Source: Research data.

Table 20 – Overall energy efficiency gains achieved by our optimised kernels compared to the baseline versions.

	auto				expl			
	novec		vec		novec		vec	
Kernel	cons	perm	cons	perm	cons	perm	cons	perm
histo	1.2×	1.4×	—	—	1.2×	1.1×	—	—
lbm_small	0.6×	1.0×	1.1×	1.1×	1.1×	1.1×	1.1×	1.4×
mri_q	1.8×	1.7×	1.9×	1.7×	1.9×	1.8×	—	—
sad	—	—	—	—	—	—	—	—
sgemm	2.9×	2.9×	2.5×	2.8×	3.3×	3.5×	3.4×	3.4×

Source: Research data.

It is possible to note that for most cases, the speedups have been similar regardless of how data packing, memory policy or `unrexp` are set. The sole exceptions are `lbm_small`

[5] Kernel frequency set to 100MHz. No pragmas are enabled, although Vivado might still perform automatic vectorisation. Banking is always enabled.

and `sad` that have cases with lower speedups or even a slowdown. For the `sad` kernel, we present the speedups as a lower bound, since the baseline kernel crashes after 13 seconds of execution. We use this value as a minimum execution time to calculate these lower bound speedups.

Table 21 presents the speedups achieved by Lina when the data transferring loops are not considered (i.e. only the computation loops are included). These values are calculated by considering the cycle count as informed by the HLS compilation, prior to full design synthesis. Although some of the kernels did not execute or fully synthesise, all kernels were at least successfully generated by the HLS step.

Table 21 – Overall speedups achieved by our optimised kernels compared to the baseline versions (computation loops only, data transfers between on and off-chip are not considered).

	auto				expl			
	novec		vec		novec		vec	
Kernel	cons	perm	cons	perm	cons	perm	cons	perm
`histo`	1.1×	1.1×	2.2×[†]	2.2×[†]	1.1×	1.1×	2.2×[†]	2.2×[†]
`lbm_small`	0.9×	2.4×	2.4×	2.4×	2.4×	2.4×	2.4×	2.4×
`mri_q`	24.2×	24.2×	18.5×	18.5×	24.2×	24.2×	85.1×[†]	85.1×[†]
`sad`	17.2×	11.5×	17.2×	17.2×	2.5×	11.5×	2.5×	2.5×
`sgemm`	10.3×	10.3×	10.3×	10.3×	10.3×	10.3×	10.3×	10.3×

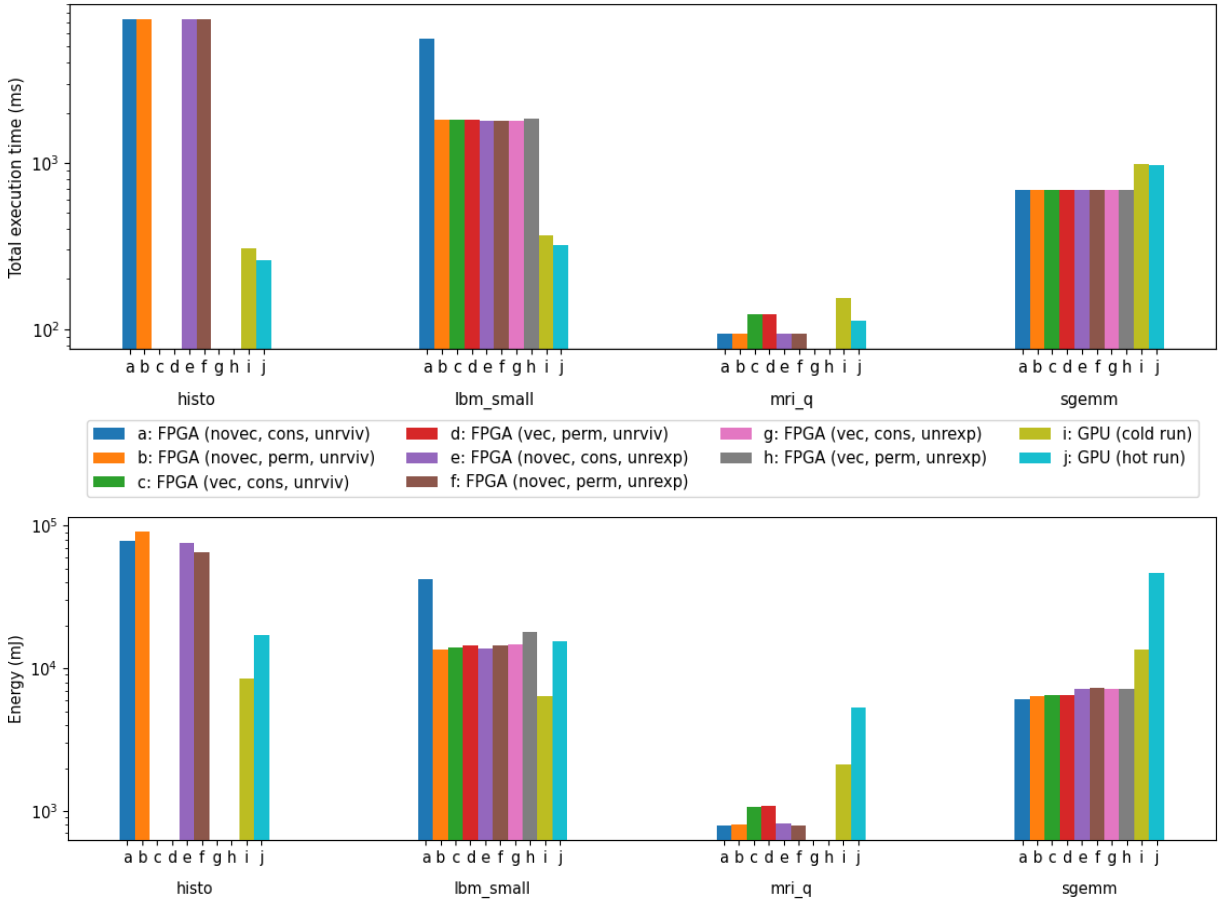
[†] HLS compilation was successful for these points, but full synthesis failed.
Source: Research data.

In general, the speedup values considering only the computation loop are greater than the overall speedups achieved. This is a strong indicator that the loops responsible for moving the data from/to off-chip memories are occupying a significant amount of the total execution time. The `mri_q` kernel, for example, presents noticeably larger speedups for the computation part than the overall, with a peak of 85.1× when `unrexp` was enabled. This was the only case where using `unrexp` brought any significant improvements, however Vivado failed to fully synthesise this point. It is also possible to note the speedup degradation that occurred when `unrexp` was enabled for the `sad` kernel.

As a final analysis, we consider the speedups from Table 21 as if they were the speedups achieved by the whole application, including data movement. This extrapolation considers a best case scenario where data movement is optimised towards little to none overhead. To calculate these values, we divide the baseline execution times from each kernel by the speedups from Table 21. Since we do not know how much energy the computation part consumes in isolation, we use the same performance speedup values from Table 21 as energy efficiency gains when considering energy consumption (i.e. we divide the consumed energy by the computation speedup). Since the speedups and energy efficiency gains both follow a similar trend on Table 19 and Table 20, such calculation

can be roughly performed. Figure 63 presents these estimated values for both execution time and energy consumption. We do not extrapolate for `sad`, since the baseline versions crashed and we could not acquire the exact execution time values. It is possible to note that the FPGA kernels still lose in both metrics for `histo` but by a lower margin. For `lbm_small`, energy consumption would be comparable. For `mri_q` and `sgemm`, the FPGA would marginally perform better, and would present an improved energy efficiency when compared to both cold and hot runs of the GPU.

Figure 63 – Extrapolated execution time and energy values for the kernels in FPGA, compared to the real values in GPU. These estimations consider that the speedup achieved by the whole FPGA application is the same as the speedup achieved by the computation loop alone.



Source: Research data.

E.3 Final Remarks

This chapter presented our framework for testing applications on FPGA and GPU. On the FPGA side, we used C/C++ kernels that were optimised by Lina. On the GPU side, we used CUDA versions of each application.

We executed and profiled five kernels from the Parboil benchmark. The GPU performs better both in execution time and energy for all kernels. The sole exception comes from the `sgemm` kernel, where FPGA had a slight better energy efficiency against the GPU but only when running in high-power mode.

Considering only the FPGA side, our optimisation approach with Lina successfully reduced both execution time and energy consumption of the applications, when compared against the baseline FPGA versions. If considering only the computation loops while ignoring the host application and data transfers, the speedup and energy improvements were even greater. This indicates that the host application and the data transfers take a significant portion of the total execution time. By considering these speedup values, we extrapolated our analysis to consider a hypothetical case where the total speedup of each application would be equal to the speedup achieved by the isolated computation parts. A similar extrapolation is performed for energy consumption. Comparing against GPU, two kernels optimised by our approach would perform better and consume less energy on FPGA.

It is also important to note that some applications consume a significant time on preparing the input data and processing the outputs. The `sgemm` kernel, for example, consumes only a small fraction of the execution time on the computation itself. Since these overheads affect both the GPU and FPGA codes, it does not affect our results presented. Nonetheless, optimising this overhead is desirable and left as future work.

It is important to note that we are comparing applications that were automatically explored for FPGA by our tool Lina, against CUDA applications that were manually coded for GPUs. In addition, CUDA is a language specifically tailored for GPUs, whereas C/C++ is an exotic input for FPGAs that requires a compiler with significant analysis and optimisation mechanisms.


ICMC
USP
SÃO CARLOS