FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# Restructuring C code for High-Level Synthesis Targeting FPGAs

**Renato Alexandre Sousa Campos**

MASTER'S DISSERTATION


Mestrado Integrado em Engenharia Informática e Computação

Supervisor: João M.P. Cardoso

Co-supervisor: João Bispo

October 22, 2020

# Restructuring C code for High-Level Synthesis Targeting FPGAs

**Renato Alexandre Sousa Campos**

Mestrado Integrado em Engenharia Informática e Computação

October 22, 2020

# Abstract

FPGAs have emerged as hardware accelerators capable of frequently being faster and/or more efficient than general-purpose hardware, such as central processing units (CPUs) and graphics processing units (GPUs). In the last decades, researchers have proposed several new languages and frameworks to allow more efficient hardware design.

This dissertation proposes a framework to automatically restructure and annotate C code with directives optimised to target FPGAs using Xilinx Vivado HLS. The inputs of the framework consist of an execution trace of an algorithm, given as a data-flow graph (DFG), and a configuration file provided by the user. We explore algorithms to generate, manipulate and optimise DFGs, such as graph pruning, reordering addition chains and isomorphic graph clustering, with the main focus set on restructuring the DFGs to efficiently exploit data-level parallelism. This approach was based on work previously developed in the *Special-Purpose Computing Systems, languages and tools* (SPeCS) research group. Our main contributions over the previous iteration include a new specification of the input DFG, improved pruning and leveling algorithms, and support for benchmarks with simple control-flow and array accesses that depend on input data. Such contributions allow us to target a new range of benchmarks.

The framework is evaluated in regards to the complexity of the output code, scalability, and synthesis results. For that, we use three benchmarks: *dotprod*, *SVM*, and *KNN*. We conclude that the code generated is not easily manually replicated due to its complexity. Furthermore, the scalability tests show that with the right configuration, it is possible to execute the backend in linear time concerning the number of nodes of the DFG. Several input sizes with multiple user configurations are synthesised for each benchmark. The experiments show that the framework is capable of generating efficient hardware implementations with significant speedups over the unmodified source codes, as is the case of the SVM, with a speedup of up to 1392×. Our best results are also competitive with other approaches proposed in the state-of-the-art.

**Keywords:** FPGA. High-Level Synthesis. Data Flow Graph. Data-Level Parallelism.

**ACM Categories:** Data flow architectures, Hardware accelerators

# Resumo

FPGAs têm surgido como aceleradores de *hardware* que conseguem frequentemente ser mais rápidos e/ou mais eficientes do que *hardware* de uso geral, tais como unidades centrais de processamento (CPUs) e unidades de processamento gráfico (GPUs). Durante as últimas décadas, foram propostas várias linguagens e ferramentas que têm como objetivo permitir um desenvolvimento mais eficiente de hardware.

Neste contexto, esta dissertação propõe uma ferramenta que gera código C restruturado e anotado com diretivas para ser sintetizado com o Xilinx Vivado HLS. A ferramenta tem como entradas um traço de execução, na forma de um grafo de fluxo de dados (DFG), e um ficheiro de configurações dado pelo utilizador. São exploradas metodologias de geração, manipulação e optimização de grafos de fluxo de dados, tais como poda de grafos, balanceamento de sequências de somas e deteção de subgrafos isomórficos, sendo o foco principal restruturar as DFGs para eficientemente expor paralelismo de dados. Esta abordagem é baseada em trabalho desenvolvido anteriormente no grupo *Special-Purpose Computing Systems, languages and tools* (SPeCS). As nossas contribuições principais sobre a iteração anterior incluem uma nova especificação da DFG de entrada, algoritmos de poda e nivelação de grafos mais eficazes, e suporte para benchmarks com controlo de fluxo simples e acessos a vetores que dependem dos dados de entrada. Estas contribuições permitem-nos processar um novo conjunto de benchmarks.

A ferramenta é avaliada em relação à complexidade do código produzido, ao tempo de execução, e aos resultados do processo de síntese. Para tal, recorremos a três benchmarks, sendo eles o *dot-prod*, o *SVM* e o *KNN*. Concluímos que o código gerado é geralmente bastante mais complexo do que o código original, e por isso, dificilmente replicado manualmente. Além disso, dado um determinado conjunto de configurações, a complexidade de execução é linear em relação ao número de nós da DFG. São testados vários tamanhos de dados de entrada e múltiplas configurações de utilizador para cada benchmark. As experiências realizadas mostram que a ferramenta é capaz de gerar implementações de hardware eficientes e competitivas com as ferramentas propostas no estado da arte, como é o caso do *SVM* em que existe uma aceleraçáo de $1392\times$ em relação ao código fonte.

# Acknowledgements

I want to express my gratitude to my supervisor, João M.P. Cardoso, and co-supervisor, João Bispo, for the guidance, revision and motivation provided. I also want to acknowledge Afonso Ferreira for establishing much of the groundwork of my dissertation and Tiago Lascasas dos Santos for helping me with the instrumentation of several benchmarks. To all the people of the SPeCS group which welcomed and took the time to help me with diverse tasks, to my family and friends, thank you!

Renato Alexandre Sousa Campos

*"The mind is not a vessel to be filled, but a fire to be kindled."*

Plutarch

# Contents

# List of Figures

# List of Tables

# Acronyms

| | |
|---|---|
| ADAS | Advanced Driver Assistance System |
| AES | Advanced Encryption Standard |
| ASIC | Application-Specific Integrated Circuit |
| ATR | Automatic Target Recognition |
| BRAM | Block Random Access Memory |
| CAD | Computer-Aided Design |
| CDFG | Control Data Flow Graph |
| CPU | Central Processing Unit |
| DCE | Dead-Code Elimination |
| DES | Data Encryption Standard |
| DFG | Data Flow Graph |
| DNA | Deoxyribonucleic acid |
| DSL | Domain Specific Language |
| DSP | Digital Signal Processor |
| FF | Flip-Flop |
| FFT | Fast Fourier Transform |
| FIR | Finite Impulse Response |
| FSM | Finite State Machine |
| FPGA | Field-Programmable Gate Array |
| GPL | General Purpose Language |
| GPU | Graphics Processing Unit |
| HDL | Hardware Description Language |
| HLL | High-Level Language |
| HLS | High-Level Synthesis |
| ICAP | Internal Configuration Access Port |
| II | Initiation Interval |
| ILP | Instruction Level Parallelism |
| IR | Intermediate Representation |
| KNN | K-Nearest Neighbors |
| LUT | Lookup Table |
| NIDS | Network Intrusion Detection System |
| PAL | Programmable Array Logic |
| PDE | Partial Differential Equation |
| RAM | Random-Access Memory |
| RC | Reconfigurable Computing |
| RC4 | Rivest Cipher 4 |
| RSA | Rivest–Shamir–Adleman |
| RTL | Register-Transfer Level |
| RTR | Run-Time Reconfigurability |
| SSA | Static Single Assignment |
| SVM | Support-Vector Machine |
| VHDL | Very High Speed Integrated Circuit Hardware Description Language |
| VHSIC | Very High Speed Integrated Circuits |
| VSFG | Value State Flow Graph |

# Chapter 1

# Introduction

## 1.1 Context

Field-programmable gate arrays (FPGAs) [68] are semiconductor devices based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. FPGAs can be reprogrammed, an arbitrary number of times, to meet application requirements. This feature distinguishes FPGAs from application-specific integrated circuits (ASICs) [86], which are manufactured for specific functionality. The hardware-reconfigurability property presents a potential performance boost and energy savings compared to software-programmable units, such as central processing units (CPUs) and graphics processing units (GPUs).

In the early days, the only way to program FPGAs was by using hardware description languages (HDLs) [34]. However, HDL writing requires expertise about hardware design. Programming in such low-level languages is very time consuming and prone to errors. As FPGAs grew in size and applications became more complex, there was a need to increase the programming abstraction. Consequently, high-level synthesis (HLS) tools [71] were developed, which allow designers to use high-level languages (HLLs) to target FPGA-based hardware. HLLs can be split into two major categories: general-purpose languages (GPLs) and domain-specific languages (DSLs) [91], [29], [24]. GPLs include languages such as C, C++, and C#, while DSLs can either be defined as completely new languages or be embedded in a GPL.

## 1.2 Motivation

Access to fast and affordable computation is of extreme importance for applications that rely on image and signal processing, cryptography, NP-hard problems, pattern matching, networking, numerical and scientific computing, amongst many others. Some industries that currently benefit from FPGA-based acceleration are the aerospace [70], medical [41] and automotive [72] ones. Therefore, softening the entry barrier for software developers to efficiently target FPGAs, will further drop the costs of using FPGAs. In turn, this will set us one step closer to a world where FPGAs are mainstream [7].

## 1.3 Problem Statement

Programming using HDLs, such as *SystemVerilog* [48] or *VHDL* [49], is time-consuming, prone to error, and requires expertise on the language and hardware being targeted. The problem is that even when using HLS tools, there is no easy way to target FPGAs without having to acquire knowledge on code optimizations and FPGAs architecture. Although HLS tools raised the programming abstraction level, now we have to manage the balance between ease of programming and performance.

## 1.4 Objectives

We intend to provide scientific research on high-level synthesis optimisations. This research includes developing and evaluating new approaches, as well as replicating and assessing compelling methodologies suggested by state-of-the-art studies. To perform such evaluations, we will develop a new framework that uses execution traces to generate C code optimised for Xilinx Vivado HLS, an idea taken from the work by Ferreira et al. [30]. We expect to develop a system that is robust and mature enough to target some real-world applications and to challenge the performance of state-of-the-art tools. We can list four major limitations of the framework developed by Ferreira et al. [30] that we plan to undertake:

- Performance - increase the performance of the synthesized code on the FPGA. The performance will be measured as a function of latency, resources used and energy costs.
- Scalability - reduce the framework execution time to make feasible to process larger program inputs.
- Control flow - handle branch conditions. This is a crucial step to be able to process a wider range of applications.
- User configurations - provide a simple interface to give the user the option to optimize different criteria, such as energy or performance.

With all of these objectives in mind, we should not forget the major one, which is to contribute to a scenario where targeting FPGAs is as easy as any other computing platform.

## 1.5 Contributions

In this dissertation, we contribute with a framework to automatically restructure software code for HLS. Figure 1.1 illustrates a simple flow of such framework. The process of instrumentation and execution collects tracing information about the program execution. The execution traces used consist of a linear listing of operations executed in a particular run of the application which represent information such as function calls, arithmetic operations, changes to variables, and branch conditions. The linear listing of operations is captured by instrumenting each statement in the

original source code to report each executed operation. In the specific case of ternary/condition operators, the trace represents all operands, even the ones which are not used. For a given input size, the trace is generic, meaning that it does not collect input data. The trace is logged in the DOT language, so it is directly represented by an acyclic DFG. This DFG is then optimized and transformed, namely to maximize data-level parallelism. The approach without considering data-level parallelism was first described in [32, 31]. Execution traces are used as they provide the real operation dependencies, which is not always true for compiler-based DFGs. However, problems such as representing control-flow and balancing runtime information vs a data-independent representation, arise when using traces. Our approach considers more complex code structures than previous work, as it is capable of dealing with simple control-flow, creating generic representations of only runtime-computable array accesses, and handling code with multiple outputs. We also contribute with algorithmic descriptions of the transformations that allow to maximize parallelism, such as reordering addition chains, isomorphic graph detection and folding of parallel subgraphs. Lastly, we provide an evaluation of the approach that shows the potential of the framework. This includes a comparison to unmodified code and to state-of-the-art results. Moreover, we study the scalability of the framework and provide evidence for its linear time complexity concerning the number of nodes of the DFG.

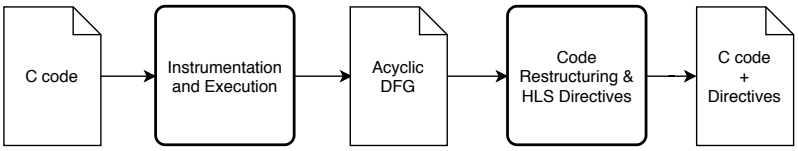



Figure 1.1: Framework flow overview

## 1.6 Dissertation Structure

This dissertation is structured as follows: Chapter 2 gives an introduction to reconfigurable systems, high-level synthesis tools and compiler optimisations. Chapter 3 provides the state-of-the-art of high-level synthesis methodologies. Chapter 4 explains the method used to approach the problem and the solution developed. Chapter 5 uses benchmarks to evaluate all aspects of the framework, from performance, scalability, and complexity of the code generated to its applicability in real-world problems. Chapter 6 discusses ideas for further work and highlights the most crucial points of this dissertation.

# Chapter 2

# Background about FPGAs and HLS

This chapter describes and explains some concepts related to FPGAs and HLS which might be useful to understand the following chapters. It provides an overview of reconfigurable systems, FPGA's architecture, compilation methods, and optimization techniques.

## 2.1 Reconfigurable Systems

According to Tessier et al. [93], Reconfigurable Computing (RC) refers to performing computation with spatially programmable architectures, such as FPGAs. To the best of our knowledge, the first FPGA was the XC2064, developed by Xilinx in 1984 [17]. FPGAs were created to mitigate the limitations of PALs (Programmable Array Logic) [94]. PALs were efficient to manufacture because their structure was very similar to memory arrays. Therefore, companies in the memory business added PALs to their lineup. Although PALs were capable of implementing custom logic, they failed to scale when the architecture grew larger to satisfy the market needs. Upon the invention of FPGAs, re-programmability was not considered an advantage, but it later proved to be a fundamental factor to reduce development costs.

### 2.1.1 Benefits of Reconfigurability

One could argue that the most important benefit of reconfigurability is the flexible reshaping of resources, which allows for mass customization of the hardware, reduction of nonrecurring engineering expenses (NRE), design rectification and iterative refinement to accommodate evolution [62]. The second benefit is that reconfigurability allows for robustness and resilience. Fault-tolerant systems, based on redundancy, can be implemented given that there are enough spatial resources in the reconfigurable fabric [55]. Another advantage is that offloading tasks to an FPGA might reduce energy consumption [88]. The last benefit of FPGAs is that they can change their functionality at run-time. This asset is called run-time reconfigurability (RTR). The benefits of RTR include having specialized hardware when required, which has to balance with the energy and space used for storing and swapping the hardware configurations. The RTR benefits have

been shown in applications such as video processing [98], human genome searching [60] and data sorting [56].

### 2.1.2 Limitations and Challenges

The challenges with reconfigurable computing lie around maximizing performance while minimizing area and energy. How can we exploit the reconfigurable architecture to program these machines and make the programming tools available to domain experts and software developers? Optimization problems that one would previously solve manually for each custom design, such as cache sizing and bit-width selection, became automation problems. There is also the challenge of how to use FPGAs as accelerators for general-purpose computers. One example of this application is a floating-point accelerator. How to manage communication between the two devices? Another challenge with reconfigurable computing is how to handle failure detection and correction. This is of extreme importance for space-deployed devices [80], due to the errors that are introduced through radiation. Thanks to partial reconfigurability, FPGAs can use their free space to perform tests for failure during operation and automatically apply corrections, extending their lifetime. A substantial part of FPGAs' success was due to advancements in Moore's law, thus there is the need to keep increasing gate density, and as such, some of the focus has been put on 3-D integration [62]. The last challenge is related to cybersecurity since it is a widespread problem in the world. As Lyke et al. [62] refer, security measures such as channel encryption, component authentication and privileged modes for self-configuring devices must be implemented in the future reconfigurable devices.

### 2.1.3 Application domains

In this subsection, we provide an overview of some of the FPGAs application domains.

One obvious application of reconfigurable hardware is **emulation of custom logic**, which might then be implemented as an ASIC.

As FPGAs grew in size, one became able to fit entire processors into a single FPGA, therefore they have been used to research **multi-core architectures** (see, e.g. [101], [75]).

**Signal and image processing** algorithms apply a set of operations repeatedly to a set of inputs. This type of operations can be highly parallelized, making them an excellent target for FPGAs. Application examples include the fast fourier transform (FFT) [52], the finite-impulse response (FIR) filter [77] and matrix-matrix multiplication kernels [54] [85]. They have also been used for image compression [35] [27] real-time face detection [64] [92] and object tracking [36].

RC is being used in the **Financial** sector to estimate prices faster than the competitors, using Monte Carlo simulations [108].

In the **Security** domain, encryption and decryption has been used to demonstrate FPGA's benefits [12]. The first demonstrations showed the performance of FPGAs on the Data Encryption Standard (DES) algorithm [61] and latter on the Advanced Encryption Standard (AES) [65] [26]. They have been used to accelerate RSA [84] [102] and to break the Rivest Cipher 4 (RC4) [95].

**NP-hard problems** usually require substantial amounts of computation compared to the problem state, making them attractive targets for FPGAs. Problems such as boolean satisfiability (SAT) [1] [76] and the travelling sales problem (TSP) [37] [63] have been successfully accelerated using FPGAs.

**Pattern matching** consists of finding patterns in large datasets, which is generally computationally intensive, but the operations are extremely regular, thus it is possible to have a computational advantage with FPGAs. Some examples of pattern matching with FPGAs include automatic target recognition (ATR) in images [99] and DNA sequence matching [42] [60] [53].

**Networking** requires handling packets at high throughput in an environment where the protocols are always evolving. FPGAs thrive in this situation because they can be reprogrammed to cope with the changes. FPGAs can be used either as routers and switches [82] [21] or as network intrusion detection systems (NIDS) [18] [40].

**Numerical and scientific computing** is dependent on the performance of floating-point arithmetic, which has been to shown to be better on FPGAs than on conventional microprocessors, at least on matrix and vector operations [97].

**Molecular dynamics** simulations involve simulating newton mechanics on thousands of particles, which is computationally demanding. FPGAs have been used to model biological molecules [6] with a 20-fold performance advantage over a conventional microprocessor.

## 2.2 FPGA's architecture

A generic representation of the architecture of an FPGA system is shown in Figure 2.1. The basic elements of reconfigurable systems are switches, building blocks, input/output blocks, configurable wiring, and a configuration management engine. Switches allow for the insertion of a configurable state. The primitive building block in FPGAs is the LUT (lookup table), which computes boolean logic. A set of LUTs is defined as a configurable logic block that can be used to implement complex functions. The other type of building block is the hard-core intellectual property (IP) which implements a well defined recurring function. Wiring connects the building blocks, thus it should be configurable to allow for flexible connections. Usually, the building blocks are not connected to all the others because this would lead to a quadratic increase in the number of wires with respect to the number of building blocks. Therefore, hierarchical wiring strategies have been developed to balance the connection degree with the performance [23]. Modern FPGAs have evolved to provide complex heterogeneous fabrics driven by market demand and technology progress. A comparison between the block diagram of an early FPGA (Spartan 3 [103]) and a modern one (Kintex 7 [104]) is shown in Figure 2.2. The configuration management engine manages the state of the reconfigurable fabric. The state is defined via a bitstream, which is a chain of bits that defines the state of each reconfigurable component. Normally, the bitstream would be transferred to the FPGA using an external configuration access port implementing the JTAG protocol, but nowadays some FPGAs can access their own configuration via the internal configuration access port (ICAP) interface.

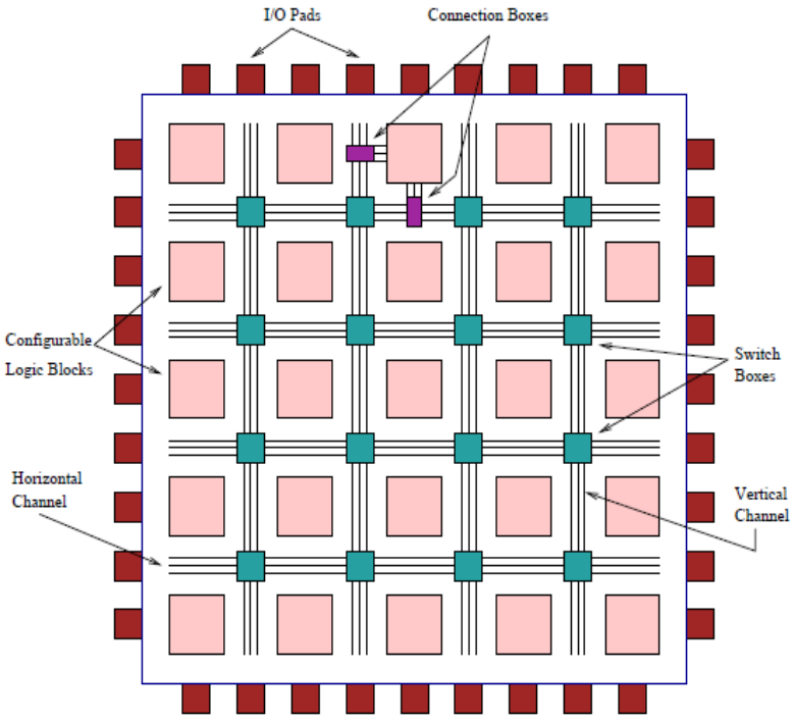



Figure 2.1: Architecture of a generic FPGA system, from Piltan et al. [79]

## 2.3 Hardware Description Languages

The process of converting a given function into an optimal set of gates is error prone and labour intensive. In the 1990s, designers realized that they would be more efficient if they raised the abstraction level. Aided by a computer-aided design (CAD) tool, designers were able to specify the logic function and automatically produce a set of optimized gates. The specifications were given in a hardware description languages (HDL). The two leading HDLs are *Verilog* and *VHDL*.

*Verilog* was developed by Gateway Design Automation in 1984 [74]. Cadence acquired Gateway in 1989 and *Verilog* was first published as an IEEE standard in 1996 [44]. It received revisions in 2001 [45] and 2006 [47]. The *SystemVerilog* [46] standard defines extensions to the IEEE 1364-2005 *Verilog* standard.

*VHDL* is a language for describing digital eletronic systems. It was developed by the US Department of Defense out of the VHSIC program that started in 1980 [5]. It became an IEEE standard in 1988 [43] and it has received several updates since.

The two major objectives of HDLs is logic simulation and synthesis. Simulation applies a set of inputs to each module and the outputs are checked to verify correct functioning. Synthesis converts each module into a netlist describing the hardware.

Using an HDL one can design circuits in different levels of abstraction. The **behavioural level** describes the order in which operations should execute. It is often not synthesizable, but it is helpful to perform logic simulation and verification. The **Register Transfer Level** (RTL) describes the hardware in more detail and defines which operations execute in each control step.

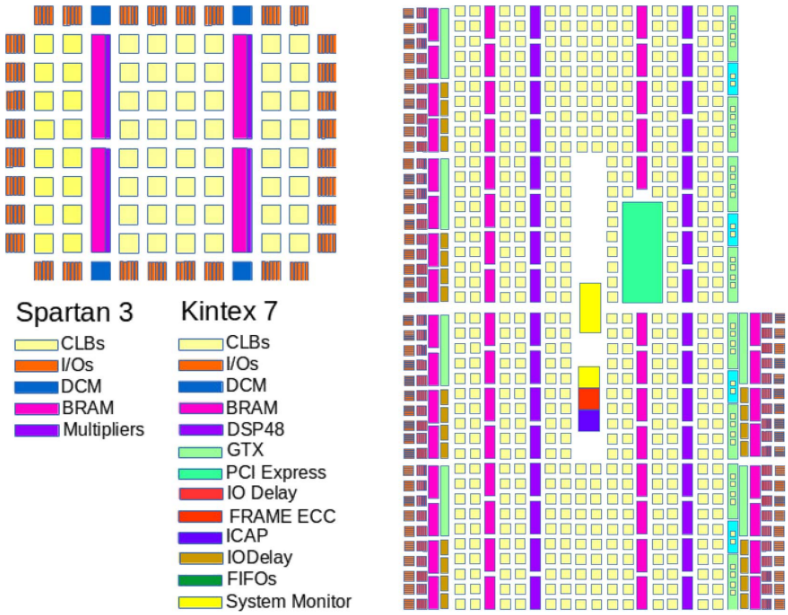



Figure 2.2: Spartan 3 vs Kintex 7 block diagrams, from Lyke et al. [62]

The lowest level of abstraction is the **gate level**, in which one describes the hardware as a set of boolean gates, but these are not assigned to the physical hardware yet.

When targeting FPGAs, additional processes (see Figure 2.3) select which physical wires and basic logic elements to use from a specific FPGA.

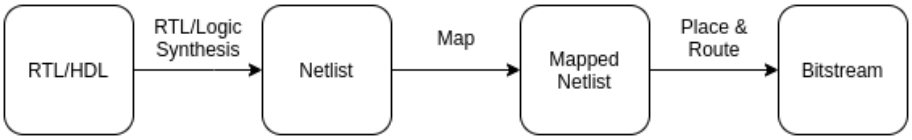



Figure 2.3: Overview of the synthesis process of an HDL into a bitstream

## 2.4 High-Level Synthesis Tools

HLS tools can synthesise a high-level language, like C, into a register transfer level (RTL) abstraction. HLS tools are designed to increase the level of abstraction in the development of applications running in hardware accelerators such as FPGAs. High-level synthesis enables designers to shorten time-to-market and to address today's complex systems. In recent years, HLS has been a popular research topic and plenty of new tools were developed, each with its optimisation techniques and input languages. A survey that compares the state-of-the-art in HLS tools, as of 2016, was published by Nane et al. [71]. Although there were a great amount of tools developed, only a few survived through time. Table 2.1 provides an overview of HLS tools still in use as of 2020.

Although LegUp HLS is a commercial tool, it started as an academic open-source tool [13] owned by the University of Toronto.

Table 2.1: Overview of HLS tools

| Compiler | Owner | Input | License | First Release | Latest Release |
|---|---|---|---|---|---|
| Vivado HLS [28] | Xilinx | C, C++, SystemC | Commercial | 2012 | 2020 |
| LegUp HLS [14] | LegUp Computing | C, C++ | Commercial | 2011 | 2020 |
| Intel HLS [90] | Intel | C++ | Commercial | 2017 | 2020 |
| MaxCompiler [89] | Maxeler | MaxJ | Commercial | 2010 | 2019 |
| Kiwi 2 [38] | U. Cambridge | C# | Academic | 2017 | 2019 |

It might be worth to highlight the key tasks involved in high-level synthesis:

**Scheduling**: Programs in high-level languages are written as a sequence of instructions, but there is no notion of timing. The scheduling task assigns computations to specific clock cycles. In other words, it creates a finite state machine (FSM) that associates each computation in the programming language with a state in the FSM.

**Allocation**: defines the number of hardware units and registers that are available to execute each operation.

**Binding**: assigns each operator and value in the programming language to a specific unit in the hardware.

**HDL generation**: After all the above tasks complete, an FSM and a data-path are defined. With this information, the HLS tool generates a circuit description in an HDL.

For a more detailed explanation about the high-level synthesis process, see [16].

### 2.4.1 Xilinx Vivado HLS

As this work uses Xilinx Vivado HLS to synthesize C code into hardware, it is important to understand some of the directives that can be applied to the code.

#### 2.4.1.1 Directives

Vivado HLS lets the users control the synthesis process through optimization directives. The tool developed explores the *ARRAY_PARTITION*, *DATAFLOW*, and *PIPELINE* directives.

When an array is defined inside of a function, Vivado HLS assign it to a BRAM, which has a maximum of 2 data ports. This limits the amount of read/write operations to 2 in each clock cycle. The problem can be solved by splitting the BRAM into multiple smaller BRAMs, increasing the number of data ports. In Vivado HLS arrays are partitioned using the *ARRAY_PARTITION* directive. Vivado HLS provides 3 styles of array partitioning: **block**, which splits the original array into equally sized blocks of consecutive elements of the original array; **cyclic**, which splits the original array into equally sized blocks of interleaved elements of the original array; **complete**, which maps each element of the original array into a register. These styles of partitioning are illustrated in Figure 2.4.

The *DATAFLOW* directive allows a series of sequential tasks to execute in parallel, thus exploiting task-level parallelism. When using this directive, tasks connect through channels, which

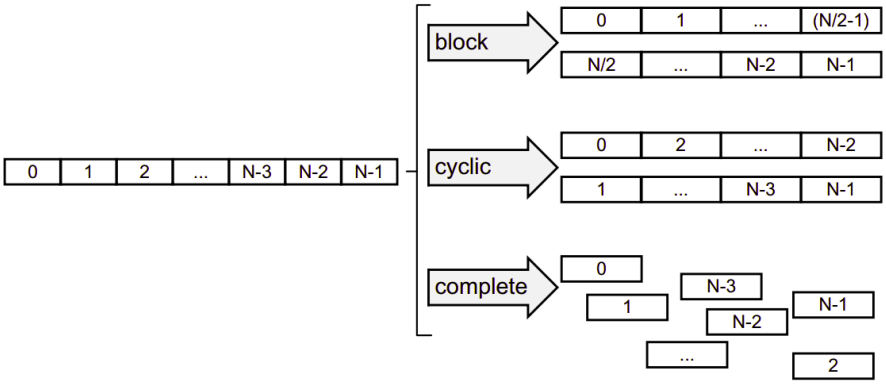



Figure 2.4: Vivado HLS array partitioning styles from Xilinx [28]

are implemented as FIFOs for scalar variables, pointers, reference parameters, and the function return. Ping-pong buffers are used for non-scalar variables like arrays [105] if Vivado HLS determines that the data is accessed in an arbitrary manner. Otherwise, if Vivado HLS determines that an array is accessed in sequential order, the memory channel is implemented using a FIFO channel with depth 1. These channels allow a task to begin before the previous task finishes and that is why the tasks overlap. This optimisation has the potential to increase throughput and reduce latency. However, there are some limitations: Xilinx recommends writing the code inside the dataflow region using the canonical form, which states that only variable declarations and function calls can be used. Moreover, each task needs to follow the single-producer-consumer model, there can not be feedback between tasks, or conditional execution of tasks, or loops with multiple exit conditions. Figure 2.5 shows that in situation B, task C effectively starts outputting the result 3 cycles before compared to situation A.

When the *PIPELINE* directive is given to a loop, it applies the software pipelining optimization.

#### 2.4.1.2 Synthesising Arrays

During initial testing of Vivado HLS, we noticed that the reported BRAM usage was dependent of array declaration and initialization. Thus, we created a synthetic benchmark (see Listing 2.1) to further investigate this issue. In this benchmark we test the effects of the const keyword, initialization and scope of the *input* array. According to the report in Table 2.2, using uninitialized global arrays or local *const* arrays seems to produce the best results. However, these are invalid, because Vivado HLS skips the operations of loading the values from the arrays. All the other configurations are valid, but declaring arrays as arguments of the top function does not use any BRAMs. This happens because parameters in the top function are by default synthesised into RTL RAM ports with the I/O protocol *ap_memory*.

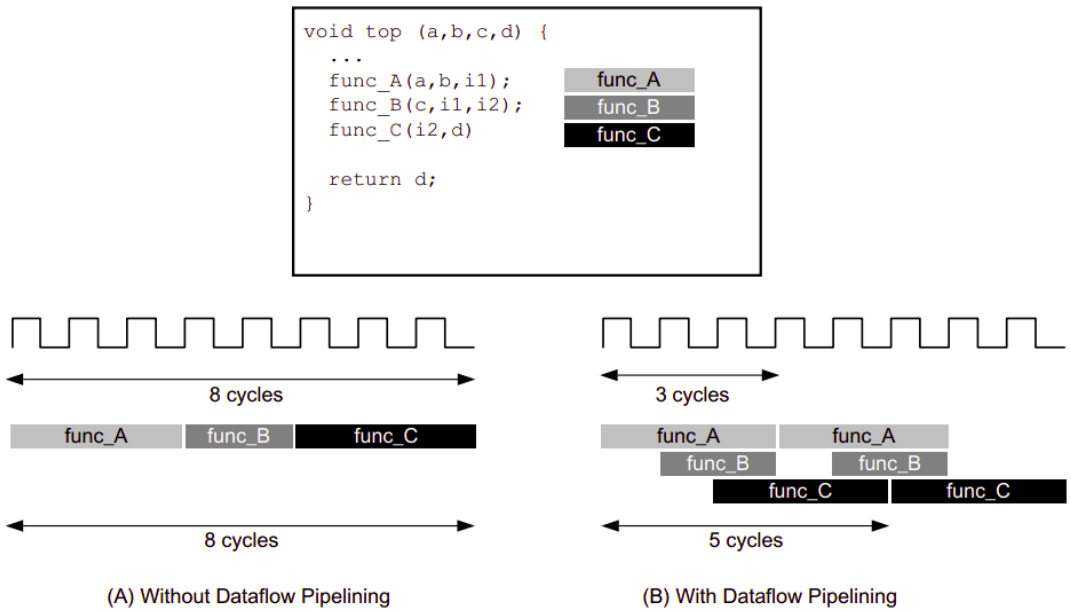



Figure 2.5: Dataflow optimization from Xilinx [105]

## 2.5 Domain Specific Languages

Domain-specific languages (DSLs) are tailored to a specific application domain and trade generality for ease of use compared to general-purpose languages (GPLs). DSL compilers can apply certain optimizations that would not be possible to do in a general domain. DSLs are not easy to develop because it requires expertise both in a specific application domain and in languages and compilers development. In spite of this fact, DSLs research and development is currently a hot topic. The following paragraph presents some state of the art DSL-to-FPGA approaches.

Watanabe et al. [100] designed an OpenACC compiler that uses the Stream Processor Generator (SPGen) DSL to target FPGAs. The SPGen DSL is being developed by the RIKEN center for computational science. Fernandes et al. [29] developed a new DSL for data analytics that can target FPGAs using pragmas. The DSL compiler developed can generate annotated C code ready to

Listing 2.1: Mini benchmark to test array declarations

```
1  #define SIZE 1024
2  int test_bram(float *avg)
3  {
4    float sum = 0;
5    for (int i=0; i < SIZE; i++) {
6      sum += input[i];
7    }
8    *avg = sum / SIZE;
9    return 0;
10 }
```

Table 2.2: Vivado HLS synthesis report for different styles of array declarations

| Scope | Const keyword | Initialized | Latency (cycles) | BRAM | Array Loads |
|---|---|---|---|---|---|
| Global | F | F | 5124 | 0 | No |
| | F | T | 7172 | 2 | Yes |
| | T | F | 5124 | 0 | No |
| | T | T | 7172 | 2 | Yes |
| Param | F | F | 7172 | 0 | Yes |
| | T | F | 7172 | 0 | Yes |
| Local | F | F | 7172 | 2 | Yes |
| | F | T | 7172 | 2 | Yes |
| | T | F | 5124 | 0 | No |
| | T | T | 7172 | 2 | Yes |

be used by Xilinx Vivado HLS. Sozzo et al. [24] developed FROST, a tool that can target FPGAs from multiple DSLs, such as Halide [81] and Tiramisu [8]. FROST takes as input an algorithm described in one of the supported DSLs and applies IR optimizations to generate HLS code. Takano et al. [91] propose a framework to synthesize Rust code into RTL. Chugh et al. [19] propose a DSL compiler to accelerate image processing pipelines on FPGAs using the PolyMage DSL. Stewart et al. [87] propose the Rathlin image processing language (RIPL), an image processing DSL for FPGAs. Fickenscher et al. [33] used the Heterogeneous Image Processing Acceleration [66] (Hipacc) DSL and compiler framework to examine the suitability of writing typical Advanced Driver Assistance System's (ADAS) algorithms in a DSL. Hipacc is a C++ embedded DSL and its compiler is capable to target Xilinx and Intel FPGAs, as well as GPUs and CPUs (see Figure 2.6). With one description of the algorithm, it was possible to generate program code for three completely different hardware architectures automatically.

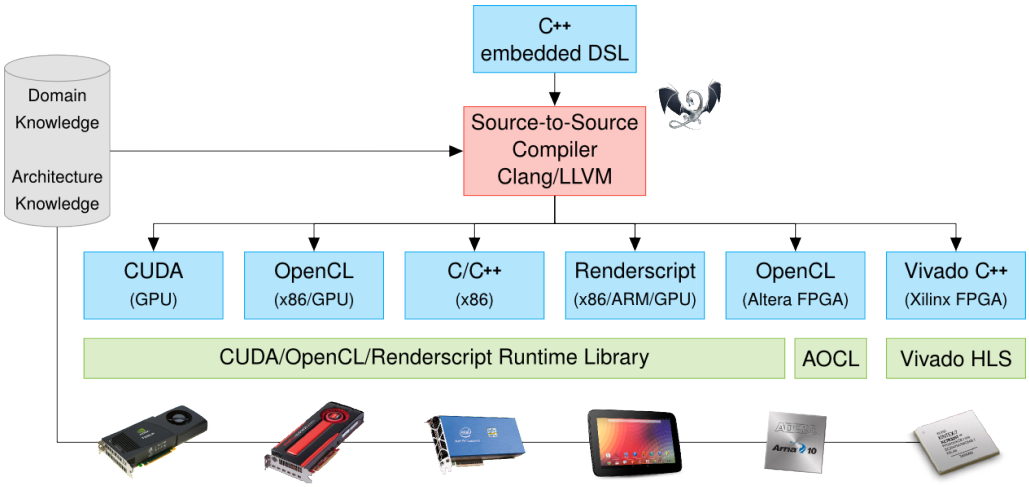



Figure 2.6: Overview of Hipacc target architectures

## 2.6 Summary

Reprogrammability is arguably FPGA's major advantage when it comes to reducing development costs. The current challenges lay on optimising their efficiency and developing user-friendly automation tools to synthesise programs into hardware. They can be used in multiple areas, like emulation of custom logic, signal and image processing, pattern matching, network and scientific computing. FPGA building blocks are the lookup tables, flip-flops, digital signal processors and block RAM, which together can be used to implement complex functions. HDLs provided a way to simulate and synthesise hardware while being less error-prone and faster to program than describing each gate. Meanwhile, HLS tools raised the level of abstraction again, allowing developers to synthesise GPLs and DSLs. Even though HLS tools are more user-friendly than HDLs, usually one needs expertise about the HLS tool to have satisfying results. The latter justifies why a considerable effort is put into researching methods to automate the generation of code targeted at HLS tools. The following chapter gives an overview of some projects that apply graph transformations that can be used to automate tasks that potentially lead to better synthesis results.

# Chapter 3

# State of the Art

This chapter presents the state of the art related to code optimizations that can be targeted at FPGAs and related to the approach proposed in this dissertation. Most of the optimizations referred in the following sections have the potential to improve execution times or to decrease area usage.

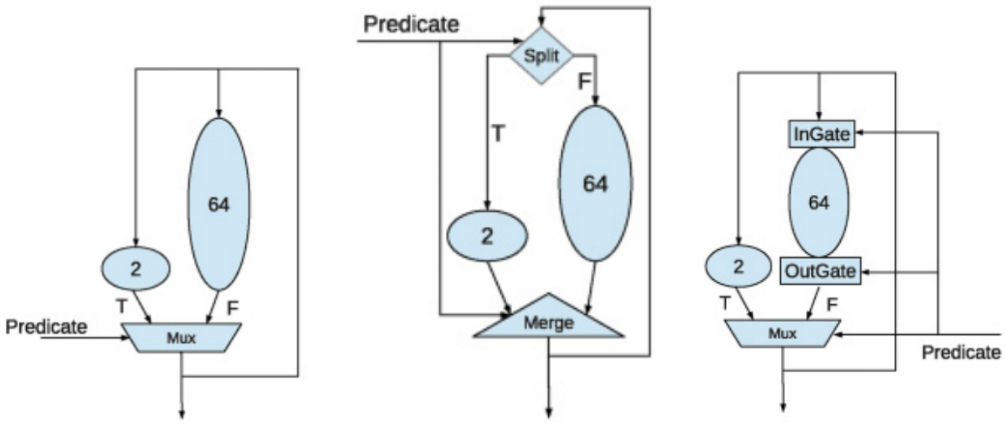
## 3.1 Value State Flow Graph: A dataflow compiler IR for accelerating control-intensive code in spatial hardware

Although reconfigurable hardware has proven to be capable of accelerating many parallel applications by orders of magnitude [107], there is a vast amount of legacy code which remains sequential. The achievable speedup is always restricted by the sequential fraction of an application. Even for applications that exhibit high parallelism, sequential performance remains essential due to constraints not considered by Amdahl's law [107], such as IO and memory bandwidth.

A new method for exploiting sequential performance is proposed by Zaidi et al. [107]. The approach consists in trying to mimic techniques already in use by conventional processors, such as aggressive branch prediction and dynamic execution scheduling. Branch prediction allows for independent instructions to be executed out of order. Dynamic execution scheduling is a method in which instructions are scheduled at run time according to the hardware resources available, in order to take advantage of parallelism which would not be visible at compile time.

Branch prediction is achieved through aggressive speculation. Figure 3.1a shows how speculation is performed by executing both data paths before the predicate is available. This may have negative consequences if the data path that takes 64 cycles does not execute regularly. The opposite of speculation is predication, which consists of executing a data path only when the predicate result is available, as shown in Figure 3.1b. The authors developed a solution called subgraph predication, in which the data path that uses 2 clock cycles is always speculated while the longer one is predicated, as shown in Figure 3.1c.

Dynamic execution scheduling was achieved by using a spatial computation model. To do so, a new compiler IR was developed, the value state flow graph (VSFG). In the VSFG, the entire program is represented as an acyclic graph, and loops are implemented as tail-recursive functions.




(a) Speculation (b) Predication (c) Subgraph predication

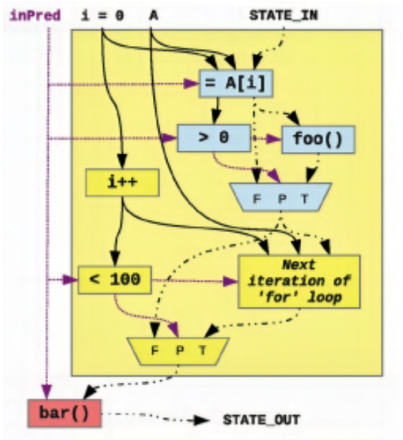
Figure 3.1: Speculation, predication and subgraph predication from Zaidi et al. [107]

It is also a hierarchical graph as all loops and function calls are represented as nested subgraphs. Such subgraphs may execute concurrently, as long as their dataflow dependencies and resources needed are satisfied. Each VSFG can be described as a labeled, directed, acyclic, hierarchical Petri net [78].

Figure 3.2 presents a VSFG of a for loop with a branch condition inside. There is no explicit notion of control flow from one block to another. The *value* edges (solid black arrows) retain the dataflow dependences from the original CDFG. The *state* edges (dashed black arrows) enforce sequentialization. The *predicate* edges (dotted purple arrows) in conjunction with multiplexers convert control flow into dataflow.

```
for (i=0; i < 100; i++)
  if (A[i] > 0)
    foo():
bar();
```

(a) Pseudo-code of a loop




(b) A VSFG from Zaidi et al. [107]

Figure 3.2: A code example and the corresponding VSFG

LLVM [58] is used to create an intermediate representation (IR). The main advantages of using LLVM are that it is possible to use any of the high-level languages supported and that the LLVM IR [58] is essentially a CDFG with some optimizations. The backend converts the LLVM IR into

a VSFG and then into Bluespec HDL [3] and Verilog HDL [48]. Bluespec is used because it is well suited for static-dataflow execution and it provides a higher level of abstraction than Verilog.

The main objective of the evaluation was to compare the performance of the VSFG and the CDFG to that of general-purpose processors on control-intensive sequential code. Cycle counts were compared against two conventional processors: an Intel Nehalem Core i7 and an Altera Nios II/f. All of the generated hardware was implemented on an Altera Stratix IV FPGA [51], using the VSFG and LegUp HLS [14] tools as backends. The LegUp HLS tool was used to provide a baseline for the CDFG performance.

Three different levels of VSFGs optimizations were tested: VSFG_0 used no loop unrolling, VSFG_1 unrolled all loops once, and VSFG_3 unrolled all loops three times. All of these implementations were configured to maximize speculative execution and used subgraph predication.

Six benchmarks were selected from the CHStone benchmark suite [106] (*dfadd*, *dfdiv*, *dfmul*, *dfsin*, *adpcm*, and *mips*) as well as two other benchmarks, *bimpa* and *epic* [11].

Although the VSFG showed to be consistently equivalent to or better than the LegUp CDFG in terms of cycle counts, the opposite is true in terms of energy consumption. VSFG_3 was capable of improving performance up to 35% over LegUp, in spite of using 3 times more energy. The cycle counts approached a simulated Intel Core i7 while using only 25% of the energy of an in-order Altera Nios IIf.

The results seem promising, and the authors refer that the following work should focus on exploiting memory architectures to improve the locality, concurrency and energy efficiency of the memory infrastructure. Another important point of research is to use code profiling to mitigate the energy needs of speculation without compromising performance.

## 3.2 A Trace-Based Approach for Code Restructuring targeting HLS for FPGAs

This section summarizes the approach by Ferreira et al. [30] [32] [31], which optimizes and restructures code targeting HLS for FPGAs, using a trace-based dataflow graph. Figure 3.3 presents an overview of the approach, which consists of the following major tasks:

1. Manually add instrumentation code to the input code. These are just print instructions that will output a DFG on execution.

2. Apply a series of optimizations: pruning, pattern matching, pipelining, reduce memory accesses, and unfold loops according to the available FPGA resources.

3. Output restructured C code annotated with the HLS tool directives.

The frontend of the framework executes the code with a given set of parameters and outputs a DFG, in the DOT language [25]. This DFG is based on a specific execution trace of the algorithm. A DFG explicitly shows the data dependencies, thus it is a natural choice when trying to improve parallelism.

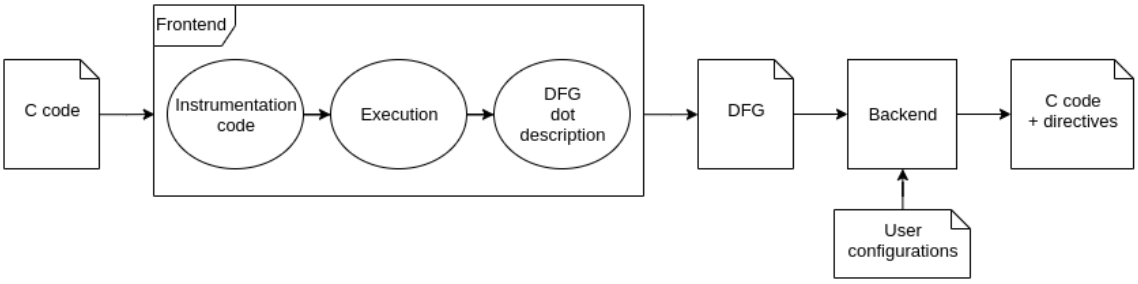



Figure 3.3: Overview of the compilation flow from [32]

The node's description is relatively simple: each node is described at least by an id, a type, and a label. A node can only be of one of three types: constant, variable or operation.

In order to output a DFG, one needs to add instrumentation code to the input code. The instrumentation rule is that one should add instrumentation code before each operation in the input code.

In practice, only C code was instrumented, but theoretically, it is possible to instrument other programming languages as well.

The backend consists of seven stages and it is responsible for restructuring and optimizing the code along with injecting the HLS directives.

The first stage is responsible for preprocessing the graph. This includes pruning of unnecessary nodes and replacing local arrays by scalar variables whenever possible. The second stage attempts to identify patterns in the DFG and returns a graph with common operations and a list of graphs of unique sequences responsible for each output. The third stage is in charge of folding the graphs output by the previous stage into a loop, compacting them into a single graph. The fourth stage implements loop pipelining on the variable that is written more often. The fifth stage improves data reuse by detecting redundant memory accesses between consecutive loop iterations. The sixth stage unfolds loops to increase parallelism, taking into account the FPGA resources made available in a configuration file. The seventh and last stage writes the DFG as C code annotated with the directives supported by the target HLS tool.

Five different benchmarks were used to evaluate the performance of the tool: the *dotproduct*, the *Autocorrelation* and the *1D fir* from the DSPLIB from Texas Instruments [50], as well as the *filter subband* from an MPEG audio encoder [15], and the *2D Convolution* provided by the UTDSP Benchmark Suite [59].

Four different optimization levels were evaluated. Level 01 applies no DFG optimizations. Level 02 applies all backend optimizations except for the fifth stage. Level 03 adds memory partitioning directives to level 02. Level 04 adds the fifth stage optimizations to level 03. These optimization levels were compared to manually optimized versions of the algorithms: C represents the original code, C-inter represents the code optimized with basic HLS directives, and the C-high improves the C-inter version with unroll and memory partition directives.

In the *filter subband* benchmark, the optimization level 03 achieved the highest speedup ($2.81\times$) as compared to the C-high version. In the *dotproduct* benchmark level 03 and 04 had

the same performance as the C-high version. For the *1D fir* benchmark the optimization level 04 had a 14.39× speedup as compared to the C-high version, showing the impact of the data reuse optimization. This is possible in the *1D fir* benchmark because 31 values can be reused from the previous iteration. The *Autocorrelation* benchmark showed that the developed tool was able to generate better unrolled loops than the unroll directive applied to the Xilinx Vivado HLS. This is true due to the fact that the unroll directive does not consider loop fusion of inner loops. The optimization level 04 was able to achieve a 7.91× speedup as compared to the C-high version. The *2D Convolution* benchmark showed that optimizing data reuse may cause imperfect loop nests, decreasing the expect speedups. However, the optimization level 04 was still able to achieve a 1.36× speedup as compared to the C-high version.

As the framework is still in its initial phase, the authors have identified several limitations:

1. The instrumentation process is manual;
2. It does not handle conditional statements yet;
3. It has scalability issues when dealing with large input traces;
4. Array accesses have to be independent of the input values;
5. It does not handle complex function calls;
6. The folding algorithm lacks flexibility;
7. There is no direct way of dealing with resource usage.

In conclusion, Ferreira et al. [32] approach has softened the entry barrier for software developers to use the computing power of FPGAs, by achieving consistent speedups even compared to manually optimized versions of the algorithms. The authors mention that the future work should focus on tackling the current limitations, by improving the flexibility and scalability of the framework.

## 3.3 Transforming Loop Chains via Macro Dataflow Graphs

This section presents the work by Davis et al. [22], which explores the use of loop chains to optimize structured grid problems on a multi-core CPU. Even though the work was not done in the context of FPGAs, there are interesting ideas in their approach which could be translated into our work.

Structured grids have been defined by Asanovic et al. [4] as one of the original seven dwarfs of computation, which means that it represents a computational kernel of many future applications. Structured grid problems can usually be solved by performing stencil computations. According to Hagedorn [39], in a stencil computation, elements of a multidimensional grid are iteratively updated. An element is updated by performing a *stencil operation* that applies a *stencil function* to a neighborhood of elements. Stencil codes may occur in specific domains, e.g. Partial Differential

Equation (PDE) solvers or image processing. As referred by Davis et al. [22], in stencil codes there are many opportunities for parallelism, but the execution time is often dominated by the time required to move large quantities of data.

Approaches to improve the performance of these applications include rewriting them using domain-specific languages (DSLs), such as PolyMage [69] and Halide [81], two DSLs developed to optimize image processing pipelines.

Loop chaining is an abstraction in which a sequence of parallel loops share data and are grouped into a chain. "In loop chaining, once an iteration of a loop completes, dependent iterations of subsequent loops are enabled to execute, rather than waiting for the entire preceding loop to complete first" [57]. A loop chain can be viewed as a set of iterations under partial ordering that makes scheduling across loops possible. Scheduling across loops enables better management of the data locality.

The work developed by Davis et al. [22] consists of optimizing stencil computations using modified macro dataflow graphs. Dataflow graphs represent data dependences, typically per statement. The modified macro dataflow graphs ($M^2$DFG) suggested by Davis et al. [22], group all iterations of a loop into a single macro node, represent data explicitly as a node and express the execution schedule.

A cost model for memory interaction is provided and transformations such as loop fusion and graph tiling are implemented. The overall objective is to minimize temporary storage requirements and to reduce storage allocations.

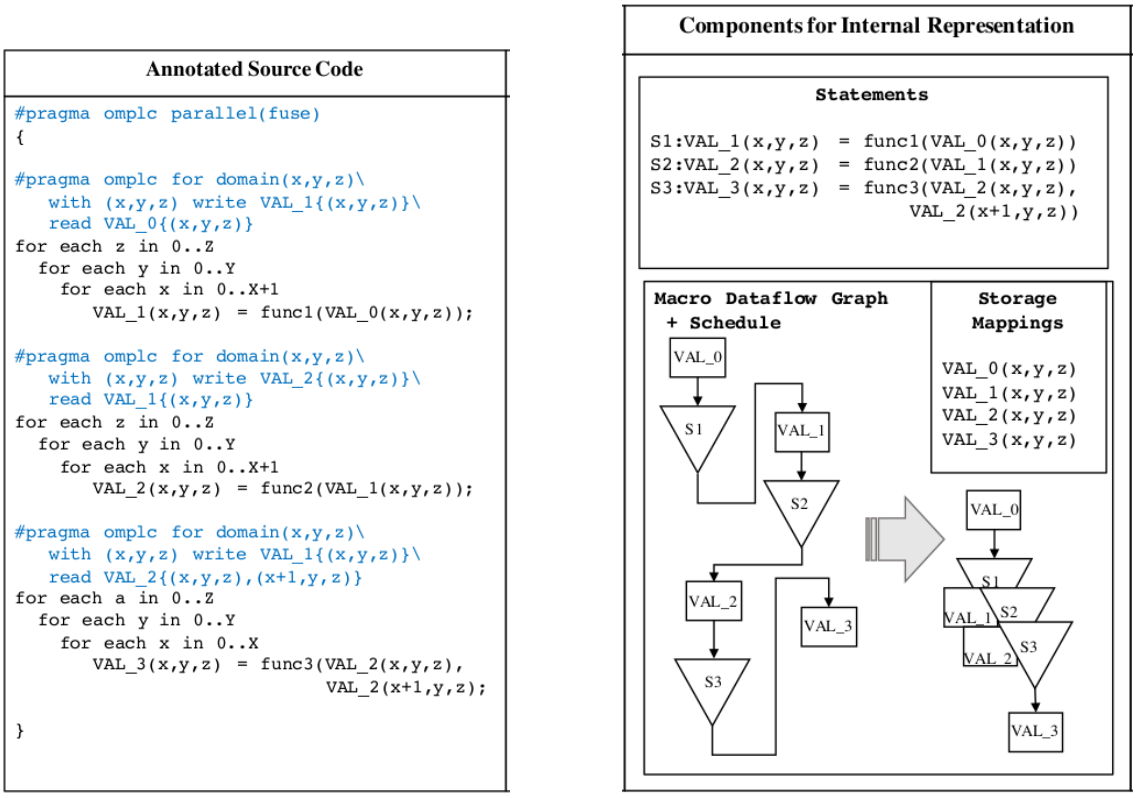
The benchmarks have to be annotated using loop chain pragmas (see Figure 3.4a). Afterward, a loop chain compiler developed by Bertolacci et al. [9] uses the annotations to apply shifts, fusion, tiling and wavefront.

Each $M^2$DFG (see Figure 3.4b) contains a set of values nodes V, a set of statement nodes S, and a set of directed edges E. A value node represents a set of values with cardinality depicted by the node's label. A statement node represents a set of expressions applied to the value sets on incoming edges. The edges of the graph express a partial execution schedule. Graphs are executed from left to right and top to bottom.

The cost model uses two metrics: the total amount of data read ($S_R$), and the maximum number of streams being accessed simultaneously ($S_C$). The transformations applied to the graph intend to reduce the $S_R$ while keeping the $S_C$ under a certain threshold.

There are 3 graph transformations defined: reschedule, and two types of loop fusion operations, producer-consumer and read reduction. Rescheduling moves a node from one row to another within the graph layout, effectively changing the execution schedule. Producer-consumer fusion merges loops resulting in a lower temporary data storage requirement. Read-reduction fusion merges loops when they read data from the same value node, effectively reducing the number of times the same data are read.

Another operation performed is graph overlapped tilling, which, unlike the previous operations, is applied to the graph as a whole. Tilling divides a problem domain into smaller subdomains called tiles. With overlapped tiles, it is possible to remove all data dependencies between




(a) Annotated Source Code

(b) Modified Macro Dataflow Graph

Figure 3.4: Overview of loop chain pragmas and the modified macro dataflow graph, from Davis et al. [22]

tiles. On the one hand, overlapped tiling has the disadvantage that some computation is repeated across tiles, but on the other hand, there is a significant increase in parallelism [73].

The last optimization consists in reducing temporary memory usage. To do it, the temporary value nodes need to be mapped into memory. The mapping depends on if a node is standalone or if it has been pulled into a statement node through fusion. Standalone nodes use a one-to-one mapping. The memory mapping for a node subject to producer-consumer fusion depends on the loop chain pragmas defined, and on the reuse distance. The memory address is provided by static liveness analysis applied to the graph.

The authors performed the experiments on the MiniFluxDiv benchmark [73], which was optimized using different schedules. Each schedule was applied to a small box of size $16^3$ and a large box of size $128^3$. The scalability of each schedule was explored by varying the thread count from 1 to 28. The experiments were conducted on a dual-socket node with an Intel Xeon E5-2680 v4 CPU with 28 cores and 192GB of RAM. Four different schedule variations were explored:

1. Series of loops: baseline implementation without any scheduling optimization.
2. Fuse among directions: performs read reduction fusion.
3. Fuse all levels: maximize both producer-consumer and read reduction fusion

4. Fuse within directions: maximize producer-consumer fusion.

For each of the previous schedules, two versions were created: a single assignment (SA) one with no storage optimizations and a version with storage optimizations (reduced). Moreover, an overlapped tiling version was applied to the Fuse all levels schedule. The overlapped tiling technique together with memory traffic reduction outperformed the state of the art embedded DSLs Halide and PolyMage. Although overlapped tiling complicates vectorization, it reduces the temporary storage needs. According to Davis et al. [22], preserving the vectorization requires an increase in temporary storage use, which scaled to 28 cores puts pressure on the memory subsystem, at least in the MiniFluxDiv benchmark.

In summary, Davis et al. [22] propose transformations on modified dataflow graphs that allow for the creation of different schedules and tilings. An algorithm for temporary storage reduction and a cost model for memory traffic are provided. The experimental results on the MiniFluxDiv benchmark show that some of the proposed optimizations outperform the achievements of the state of the art DSLs.

## 3.4 Using graph isomorphism for mapping of data flow applications on reconfigurable computing systems

Finding two similar graphs is a problem called the graph isomorphism problem. The complexity of this problem is not know, however there are algorithms that solve it for most of the input graphs in polynomial time. Mishra et al. [67] propose an algorithm to find isomorphic subgraphs in polynomial time for dataflow graphs and interface them as hardware accelerators in the system-on-chip design flow. Although with a different objective, in this dissertation we base the identification of data-parallelism in their algorithm to find isomorphic subgraphs. It works by computing the weight of each graph, which is like computing an identity for each one. Then, graphs with the same weight are clustered together and labeled as isomorphic.

## 3.5 Summary

This chapter presents some state-of-the-art approaches of compilation techniques that may be applied to restructure code for HLS. In control-intensive code, subgraph predication and dynamic execution schedule can be used to gain a performance advantage, as shown by Zaidi et al. [107]. Another promising approach to restructure code to HLS uses execution traces and applies DFG optimisations to generate code with directives [30] [32] [31]. Its main limitations are related to scalability for large input traces and the number of C features supported. Also, Davis et al. [22] show that it is possible to use loop chains to optimise stencil computations. Even though the target was a multi-core CPU, the idea could be applied to FPGAs. Lastly, Mishra et al. [67] describe an efficient method to detect and cluster isomorphic graphs in polynomial time.

# Chapter 4

# Description of the Framework

The framework developed takes an algorithm in C code and outputs code optimized for Vivado HLS. The flow of this process is illustrated in Figure 4.1. The first step is to instrument the C source code and execute it to generate a DFG in the dot language. The instrumentation process is partially automatic thanks to Santos et al. [83]. Given a DFG in the dot language and a JSON configuration file provided by the user, the backend applies a series of algorithms that restructure and optimize the DFG. Lastly, the backend uses the DFG to write C code optimized for Vivado HLS.

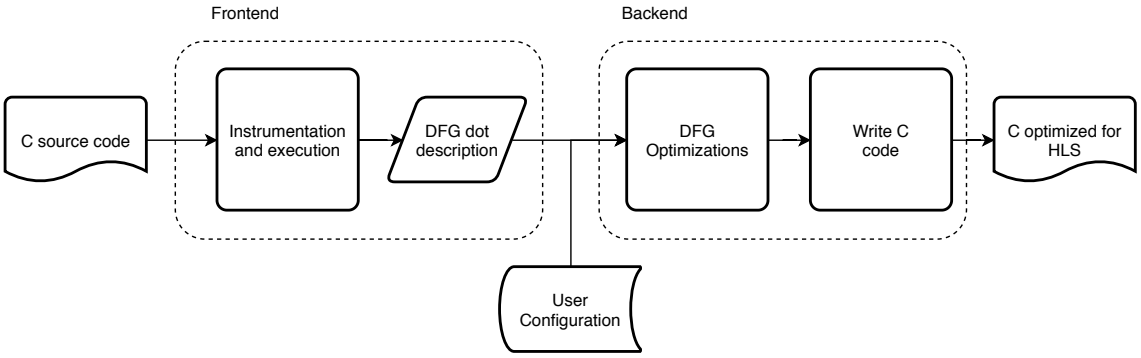



Figure 4.1: Framework flow overview

## 4.1 DFG DOT description requirements and specification

The objective of the frontend is to generate a DFG that represents an algorithm execution. A brief description of the traces used is given in Section 1.5. This DFG should contain all the operations of a given algorithm and each edge represents a data dependence. In addition, the DFG has to be a direct acyclic graph. If a cyclic graph is given to the backend, the leveling algorithm will detect it and halt execution. In the DOT language a direct graph is created using the "Digraph" keyword. The DFG should be written in the SSA form, meaning that each time a variable is written, a new node should be created for it. The SSA form is used because it improves and eases a variety

of compiler optimizations, such as dead-code elimination and constant propagation. The task of creating the DFG can be done manually or automatically, through the insertion of print commands in the original code. At the moment, the types of nodes supported at the input DFG are described in Table 4.1.

Table 4.1: Description of the nodes that can be used in the input DFG.

| **Node Type** | **Attributes** |
|---|---|
| Operation | label: symbol of the operation<br>att1: op |
| Variable | label: name of the variable<br>att1: var<br>att2: scope of the variable<br>att3: type of the variable |
| Constant | label: value of the constant<br>att1: const |
| No Operation | att1: nop |
| Multiplexer | att1: mux |
| Assignment | att1: assignment |
| Complex Assignment | att1: complexAssignment |

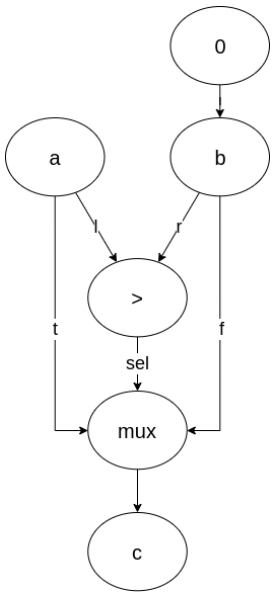



Figure 4.2: DFG representing $c = (a > b)\ ?\ a : b$

```
1  Digraph {
2    const_0 [label=0, att1=const];
3    a_0 [label=a, att1=var, att2=param, att3=int];
4    b_0 [label=b, att1=var, att2=loc, att3=double];
5    op_0 [label='>', att1=op];
6    mux_0 [att1=mux];
7    c_0 [label=c, att1=var, att2=loc, att3=double];
8    const_0->b_0 [];
9    a_0->op_0 [pos=l];
10   b_0->op_0 [pos=r];
11   op1->mux_0 [pos=sel];
12   a_0->mux_0 [pos=t];
13   b_0->mux_0 [pos=f];
14   mux_0->c_0 [];
15 }
```

Listing 4.1: DOT description of the DFG in Figure 4.2.

### 4.1.1 Data reads and writes

At the frontend, edges represent data dependencies, thus an edge that enters a node carries a dependency from the previous node. As an example, in Figure 4.2 the edges from *a* and *b* to the ">" node represent data reads, while the edge that leaves the operator represents a data write into the *mux* node. In the DOT language [25], directed edges are represented by the arrow symbol (->). For example, line 9 of the listing above, represents an edge from the node a_0 to the node op_0, with the attribute 'pos' set to 'l' (left).

### 4.1.2 Arithmetic operations

Arithmetic operations such as additions, subtractions, divisions, multiplications, shifts, and comparisons, can be represented by nodes that have their labels equal to the symbol of the operation and their attribute "att1" equal to "op". Each operation should be represented by a node with a unique ID, otherwise the data dependencies may be incorrect. Each operation has 2 operands represented by two entering edges, and each edge needs to have the "pos" attribute equal to "l" or "r", meaning "left" or "right", to ensure a correct order. The "pos" attribute is used by the backend to compare edges, thus even sum operations should have the "pos" attributes defined. Each operation should have a single output to avoid doing the operation multiple times.

### 4.1.3 Variable nodes

Each time that a variable node is written, a new node with a different ID should be created for it. The label attribute refers to the name of the variable in the source code. The attribute "att1" refers to the type of node, so this should always be set to "var". There are 3 scopes defined: "global", "param" or "inte", and "loc". The scope "global" refers to global variables, that are defined outside of the top-level function. "param" or "inte" refer to variables that are arguments of the top-level

function. They refer both to the same thing. "inte" is only supported for legacy reasons, and it is short for "interface". "param" is short for "parameter". The last scope available, "loc", stands for "local" variables.

### 4.1.4 Constant nodes

Constant nodes hold their value in the "label" attribute. The "att1" field should always be set to "const".

### 4.1.5 Ternary / conditional operators

Ternary operators are a simple form of conditional expressions. In C, a ternary operator $a$ ? $b$ : $c$, where $a$, $b$, and $c$ can be expressions, evaluates to $b$ if $a$ is true, and otherwise to $c$. A DFG that represents a ternary operator is shown in Figure 4.2. The node that allows for ternary operators to be represented in the DFG is the *mux* node. This node needs to have exactly three entering edges, with the "pos" attributes set to "sel" (selector), "t" (true), and "f" (false) respectively. The output of a "mux" node should be a variable that will store the result.

### 4.1.6 No operation node

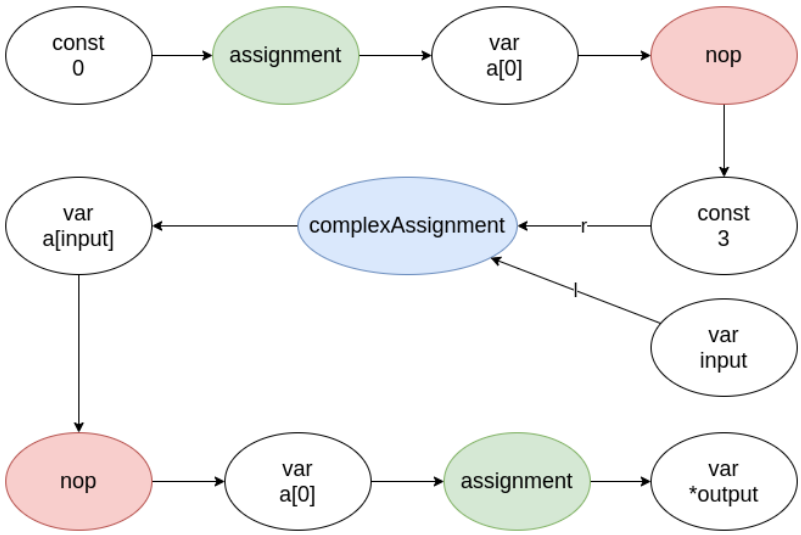



Figure 4.3: DFG representing Listing 4.2.

The "nop" node can be used to represent data dependences that are not defined otherwise. For example, in Listing 4.2 the "nop" node is needed to ensure that the operation "a[input]=3" executes only after "a[0]=0" and that "*output = a[0]" executes only after the previous two instructions. The "nop" node is only needed because the array "a" is accessed at an index that is unknown at compile-time.

```
1 void f(int input, int a[10], int* output) {
2   a[0]=0;
3   a[input]=3;
4   *output = a[0];
5 }
```

Listing 4.2: C code example that justifies the need for the "assignment", "complexAssignment" and "nop" nodes.

### 4.1.7 Assignment node

The assignment node is used to guarantee that the variable written is not going to be pruned by the backend. This is needed in situations where there are array accesses that can only be computed at run-time or to ensure that an output variable gets written. As an example, if we represent the statement "a[0]=0" from Listing 4.2 as a simple edge from a "const" node to a "var" node, the pruning algorithm will effectively replace all of the "a[0]" data reads by the constant 0. In this situation this would be incorrect, because the statement "a[input]=3" may change the value of "a[0]".

Thus, to represent the code in Listing 4.2 we need to use the assignment node with an entering edge coming from the "0 const" node and a leaving edge going to the "a[0] var" node, as illustrated in Figure 4.3. Notice also the use of the assignment node to represent the statement "*output = a[0]". This is needed to prevent the backend from making any invalid optimizations in this statement.

### 4.1.8 Complex Assignment node

The "complexAssignment" node should be used when the left-hand side of an assignment statement contains a dependency that is unknown at compile time. This is exemplified by the statement "a[input]=3" from Listing 4.2. The "complexAssignment" node has always two entering edges, one that represents the left-hand side dependency ("pos" attribute set to "l") and other that represents the right-hand side of the assignment ("pos" attribute set to "r"), as illustrated by Figure 4.3.

### 4.1.9 Calls to functions

Calls to functions with a single parameter are supported through the use of the "mod" attribute in a node. As an example, the DOT description that represents the C code "a = sqrt(b)", should be written as in Listing 4.3. This syntax was carried over from the work by Ferreira et al. [30], but we reckon that a more generic approach should be used in a future version.

```
1 Digraph {
2 a_0 [label=a, att1=var, att2=param, att3=int];
3 b_0 [label=b, att1=var, att2=param, att3=double];
4 a_0 -> b_0 [mod='sqrt(']
5 }
```

Listing 4.3: Use of the mod attribute

## 4.2 Frontend limitations

There is no syntax defined for calls to functions with more than one parameter. Thus, it is not possible to instrument a call to $func1(a,b)$, but it is possible to instrument $func2(a)$ using the *mod* attribute. This is a restriction of a particular representation and not a restriction of our approach. It is also important to notice the disadvantages inherited by using a DFG that is a full trace of execution: when the benchmarks generate graphs with millions of nodes, which happens when there are cycles that iterate millions of times, it becomes infeasible for any algorithm to parse and transform these. The last limitation is related to the fact that "if" and "switch" constructs are not supported directly, thus one needs to manually transform these into ternary/conditional operations, which might not always be feasible.

### 4.2.1 Information lost through tracing

A downside of the fact that the input DFG represents a trace of execution, is that if there are kernel parameters which control the flow of execution, then the DFG is only correct for that combination of parameters.

## 4.3 User configuration

The user configuration is a JSON file which allows to control backend execution. At the moment of writing, there are 13 optional and 6 mandatory configurations. The SVM benchmark (see Listing A.1) is used to give examples for each configuration. A full example of a configuration file for the kNN benchmark is available in Listing C.1.

### 4.3.1 Mandatory configuration options

The mandatory configurations are the following: "inputs", "input_types", "outputs", "output_types", "graph", and "outputFile". An example of these configurations being used in the SVM benchmark is shown in Table D.1. The remaining options have defaults values defined. The optional configurations list includes: "fold", "parallelizeSums", "arithmetic", "pruneLocalArrays", "saveEnergy", "parallelFunctions", "maxNodesPerSubgraph", "subgraphRepeats", "minFoldLevels", "maxFoldLevels", "varsToPartition", "includes" and "defines". Syntax examples and a brief description for each option is shown in Table D.2.

## 4.4 Backend stages

This section explains the transformations that the DFG goes through in the backend. The algorithms applied are hardcoded into a transformation queue because for the benchmarks tested there was no need to provide a different execution order. However, in the future this queue might be provided by the user. At the moment there are six main stages defined, but the code might be extended to add new ones. Figure 4.4 shows the complete flow of execution of the backend.

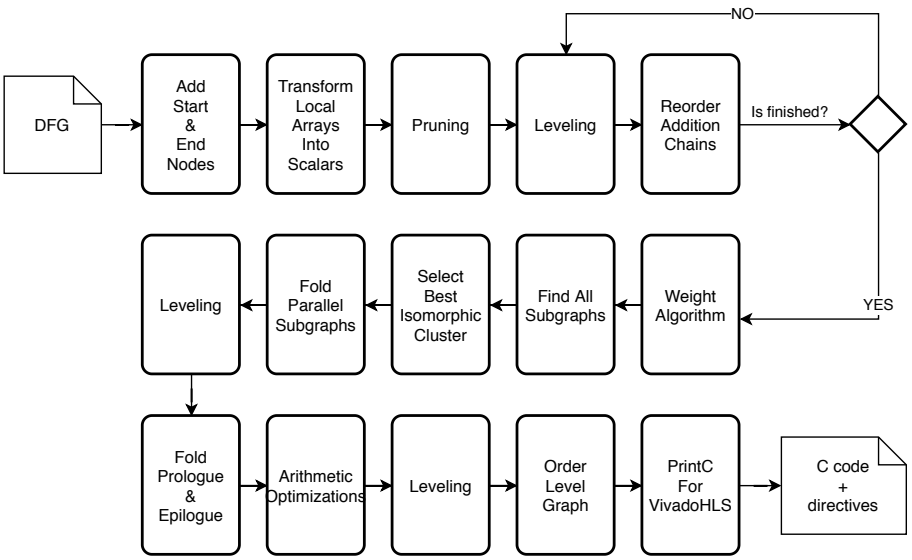
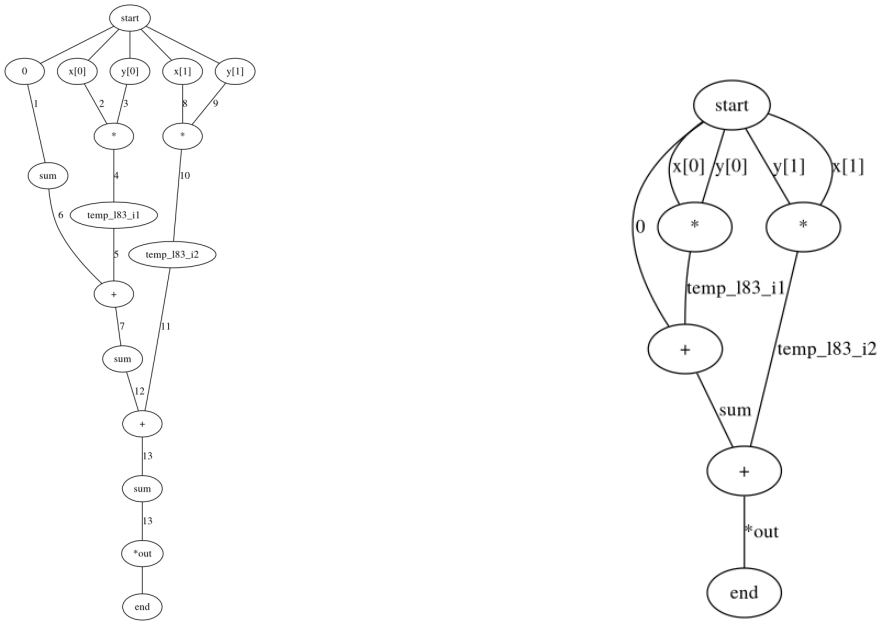



Figure 4.4: Backend execution flow.

Stage 1 prunes and simplifies the DFG. Stage 2 reorders addition chains to increase operation-level parallelism. Stage 3 finds clusters of nodes that can be executed in parallel. Stage 4 folds the parallel clusters found in Stage 3. Stage 5 wraps the nodes that were not folded into new graphs. Stage 6 applies arithmetic optimizations. All the stages depend on the output of the *Pruning* algorithm because it modifies the structure of the graph significantly. In order to ensure the extensibility of the code, each stage was designed with no assumptions about the input graph, except for the structure that is defined in Subsection 4.4.1. The *Dotprod* benchmark (see Listing A.2) is used as an example to help describe the algorithms.

### 4.4.1 Pruning

The pruning algorithm removes redundant information and moves data from nodes into edges, reducing the graph complexity. Nodes that are variables or constants are selected for replacement. Of these nodes, if one is between two op nodes, it is removed and replaced by an edge containing the information removed - e.g., the node *sum* that is between edges 7 and 12 in Figure 4.5a is removed and replaced by an edge containing the *sum* variable (see Figure 4.5b). Nodes that are disconnected from the graph are also removed. If a constant is being assigned to a variable, we replace both the constant and the variable nodes by a single edge - e.g., in Figure 4.5a the constant




(a) Before pruning.

(b) After pruning.

Figure 4.5: Pruning execution (see Section 4.4.1) on a DFG representing the *Dotprod* with $N = 2$.

node *0* is an input to the variable node *sum*, but in Figure 4.5b both have been replaced by an edge from the *start* node into the + operation node, containing the constant data.

**Input:** graph
**Output:** Simplified graph
1: **for each** *node* ∈ *graph* **do**
2: **if** ISDISCONNECTED(node) **then** REMOVE(node)
3: **else if** ISVAR(node) OR ISCONST(node) **then** replaceByEdge(node)
4: **end if**
5: **end for**

Figure 4.6: The pruning task.

### 4.4.2 Leveling

Some graph transformations require knowing the level of each node. We define the level $l$ of the "Start" node to be $l = 0$. Then for each other node $N$, its level $N_l$ is given by:

$$N_l = max(N.parents_l) + 1 \tag{4.1}$$

where $N.parents_l$ corresponds to the levels of all parent nodes of $N$. A consequence of Equation 4.1 is that a node can only be leveled when all of its parents are already leveled. Our *Leveling* algorithm is described in Figure 4.7.

**Input:** A DFG *graph*
**Output:** a *graph* with leveled nodes and a *levelGraph* containing the lists of nodes in each level.

```
1:  nodesLeveled ← ∅
2:  levelGraph ← ∅
3:  StartN ← graph.get("Start")
4:  LEVELNODE(StartN, 0, levelGraph)
5:  nodeSet ← GETLEVELABLECHILDREN(StartN)
6:  level ← 0
7:  while nodeSet ≠ ∅ do
8:      level ← level + 1
9:      nodesToAdd ← ∅
10:     for each N ∈ nodeSet do
11:         LEVELNODE(N, level, levelGraph)
12:         nodesLeveled ← nodesLeveled ∪ {N}
13:         nodesToAdd ← nodesToAdd ∪ GETLEVELABLECHILDREN(N)
14:     end for
15:     nodeSet ← (nodeSet \ nodesLeveled) ∪ nodesToAdd
16:     nodesLeveled ← ∅
17: end while
18: return levelGraph
```

Figure 4.7: The leveling algorithm.

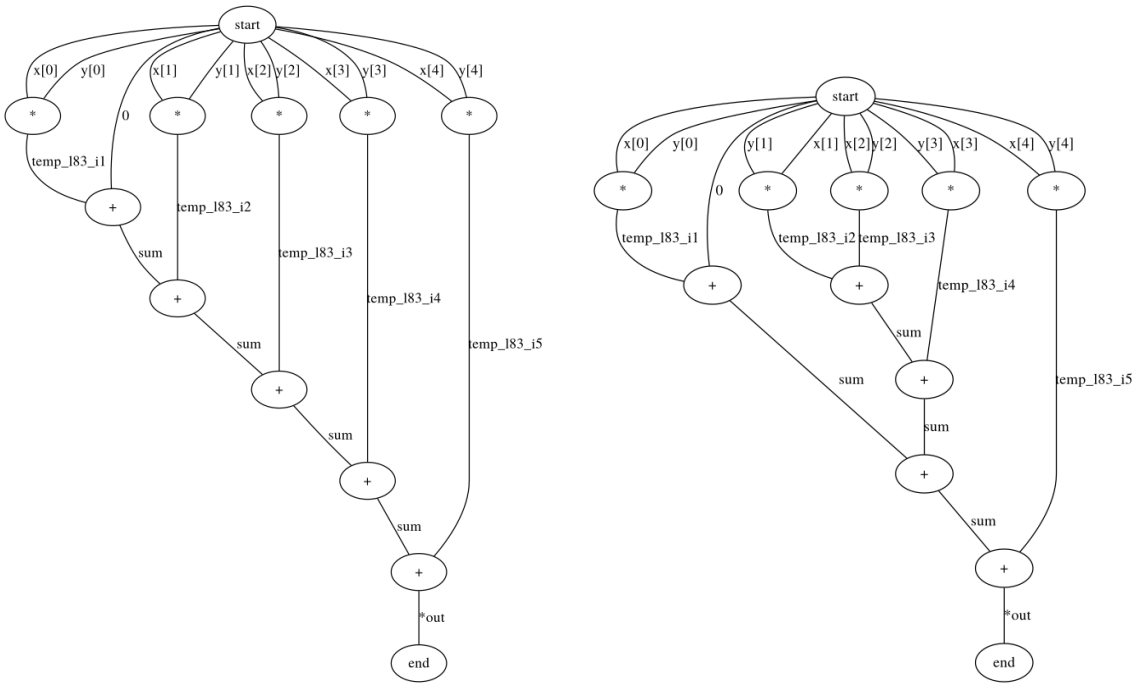
### 4.4.3 Balancing addition chains

The core transformations that allow extraction of data-level parallelism start with an algorithm that balances addition chains to increase operation-level parallelism and potentially unlock data-level parallelism, which was not clear otherwise. In the case of the *Dotprod* benchmark, this transformation affects the order in which the multiplication results are added, reshaping the addition process from an operation chain into a triangle-shaped subgraph. As an example, in Figure 4.8a there is a sequence of five additions and after balancing we obtain the graph represented by Figure 4.8b.

The *BalanceAdditionChains* 4.9 algorithm iterates through all graph levels and searches for addition chains starting at each level using the *DetectChains* 4.10 algorithm. If there are addition chains starting at a given level, then rotations are applied, using the algorithm described in Figure 4.11, to maximize the number of addition operations in that level.

### 4.4.4 Isomorphic Matching

This stage is responsible for finding clusters of nodes that can be executed in parallel. The algorithm works very similar to the one presented by Mishra et al. [67] as it gives a weight to each node, computes a list of subgraphs and finally matches subgraphs that have the same weight.




(a) Before balancing

(b) After balancing

Figure 4.8: Balancing addition chains on the *Dotprod* DFG with $N = 5$.

**Input:** a *graph* to apply the transformation.
**Output:** a *graph* with reordered addition chains.

1: $additionChains \leftarrow \varnothing$ ▷ List<LinkedList>
2: $LG \leftarrow graph.levelGraph$
3: **for** $level = 1; level < graph.maxLevel; level \leftarrow level + 1$ **do**
4: $additionChains \leftarrow$ DETECTCHAINS$(LG, level)$
5: **if** $additionChains \neq \varnothing$ **then**
6: ROTATEGRAPH$(additionChains)$
7: LEVELGRAPH$(graph)$
8: **end if**
9: **end for**

Figure 4.9: Balance Addition Chains algorithm

**Input:** leveled graph *levelGraph*, level in which addition chains start *minLevel*.
**Output:** a list where each element is a linkedlist of addition nodes.

```
1:  chains ← ∅                                                    ▷ Hashmap
2:  chainHeads ← ∅                                                ▷ Hashmap
3:  nodesToAnalyze ← ∅                                            ▷ LinkedList
4:  for each node ∈ levelGraph.at(minLevel) do
5:      if ((node.isSum)) then
6:          chains.add(node, {node})
7:          nodesToAnalyze.add(node)
8:      end if
9:  end for
10: while nodesToAnalyze ≠ ∅ do
11:     parent ← nodesToAnalyze.head
12:     chainHead ← chainHeads.parent
13:     for each child ∈ parent.children do
14:         if (child.isSum) then
15:             chainHeads.put(child, chainHead)
16:             chains.get(chainHead).add(child)
17:             nodesToAnalyze.add(child)
18:         end if
19:     end for
20: end while
21: return chains.values.filter(chain.size > 3)
```

Figure 4.10: Detect Addition Chains algorithm

**Input:** a *graph* conatining the *additionChains*
**Output:** a *graph* with reordered addition chains.

```
1: for each chain ∈ additionChains do
2:     while chain.size ≥ 4 do
3:         MOVEEDGE(secondNode,"r",thirdNode,"l")        ▷ Move the edge with attribute
   pos="r" that points to the second node in the chain, to point to the third node and set pos="l"
4:         MOVEEDGE(fourthNode,"l",secondNode,"r")
5:         MOVEEDGE(thirdNode,"l",fourthNode,"l")
6:         chain.removeAtIndex(2)
7:         chain.removeFirst()
8:     end while
9: end for
```

Figure 4.11: Rotate Graph algorithm

#### 4.4.4.1 Weighting Algorithm

To compute the weight of each node we use hash tables that map each supported operation, node types, edge types and edge positions to integer values. Thus, we defined the weight of a node N, $W_N$ as:

$$W_N = k_1 \cdot N.level + (k_2 + n.degree) \cdot N.typeW + \sum_{E \in N.inEdges} W_E \tag{4.2}$$

, where $k_1$ and $k_2$ are constants, $N.typeW$ is the weight of the node type, $N.inEdges$ are the edges that enter node $N$, and the weight of an edge E, $W_E$, is defined as:

$$W_E = E.typeW + E.positionW + E.nameW + E.dim \tag{4.3}$$

, where $E.typeW$ is the weight of the edge type, $E.positionW$ is the weight of the position label, and $E.nameW$ is an integer hash code of the variable name modulo by a constant. $E.nameW$ and $E.dim$ are 0 if E is not an array.

#### 4.4.4.2 All Subgraphs Algorithm

The *all subgraphs algorithm* finds all the subgraphs in a given graph. In graphs which have many levels this is the algorithm that takes most time to execute, because its average complexity is $\mathcal{O}((l^2 - l) \cdot avg(n_l))$, where $l$ is the number of levels in the graph and $n_l$ is the number of nodes per level. The pseudo-code for the algorithm is available in Figure 4.12. As it can take a while to execute this algorithm for large graphs, we give the user the option to limit the minimum and the maximum levels of the subgraphs. While building each temporary subgraph, the subgraph weight is computed as:

$$subgraphWeight = \sum_{node \in graph} node.weight \tag{4.4}$$

Therefore, subgraphs can be grouped by weight immediately after building each one. The idea is that subgraphs with the same weight can be executed in parallel. Figure 4.13 and Table 4.2 depict the groups of subgraphs detected in this stage.

Table 4.2: Dotprod parallel subgraphs for N=10

| Color | # Subgraphs | # Subgraph Nodes | # Subgraph Levels | #Subgraphs * #Subgraph Nodes |
|---|---|---|---|---|
| Red | 10 | 1 | 1 | 10 |
| Green | 5 | 1 | 1 | 5 |
| Purple | 2 | 1 | 1 | 2 |
| Blue | 4 | 3 | 2 | 12 |
| Pink | 2 | 7 | 3 | 14 |

**Input:** leveled graph $LG$
**Output:** isomorphic subgraphs $IS$ mapping weights to a list of subgraphs

```
1:  IS ← ∅                                              ▷ Implemented as an HashMap
2:  for i = 1; i < maxLevel; i++ do
3:      for j = i; j < maxLevel : j++ do
4:          NodesInS ← ∅                                ▷ Implemented as an HashSet
5:          for each N ∈ LG.at(i) do
6:              if (N ∉ NodesInS) then
7:                  S ← addNode(N, NinS, i, j)   ▷ Adds node N and all of its direct and indirect
    connections between levels i and j
8:                  if (S.maxLevel == j) then
9:                      IS.add(S.weight, S)
10:                 end if
11:             end if
12:         end for
13:     end for
14: end for
15: return IS
```

Figure 4.12: Algorithm to Find All Subgraphs.

### 4.4.5 Folding parallel subgraphs

The first step of the folding algorithm is to select which subgraphs are to be folded. This is done by using an heuristic that evaluates each subgraph. The current heuristic, $h(s)$, is defined as follows:

$$h(s) = s_p \times n_s \tag{4.5}$$

$$s_p > 2 \tag{4.6}$$

It takes into account only the number of parallel subgraphs, $s_p$ and the number of nodes in each subgraph $n_s$. Therefore, it maximizes both the number of parallel subgraphs and the size of each subgraph. Constraint 4.6 is needed to compute the arithmetic and geometric progressions of each edge. Since in the FPGA's realm we are always constrained by the available resources, the framework lets the user choose the maximum number of nodes in each subgraph. By reducing the number of nodes in each subgraph the complexity of the functions that are to be parallelized decreases. In spite the fact that reducing the complexity of the subgraph usually leads to a larger number of parallel subgraphs, the area used is effectively reduced because the number of parallel calls is a constant defined by the user. Increasing the number of parallel subgraphs leads only to a larger number of iterations of the parallel functions.

In Table 4.2, the pink group maximizes the selection heuristic. However, it is invalidated by Equation 4.6. Therefore, the blue clusters are selected to be folded. The output of the folding stage consists of two graphs. The main graph (see Figure 4.14a) represents the top-level function, while the other one (see Figure 4.14b) represents the function that may be called in parallel.

The algorithm that folds parallel subgraphs is outlined in Figure 4.15. To represent calls to

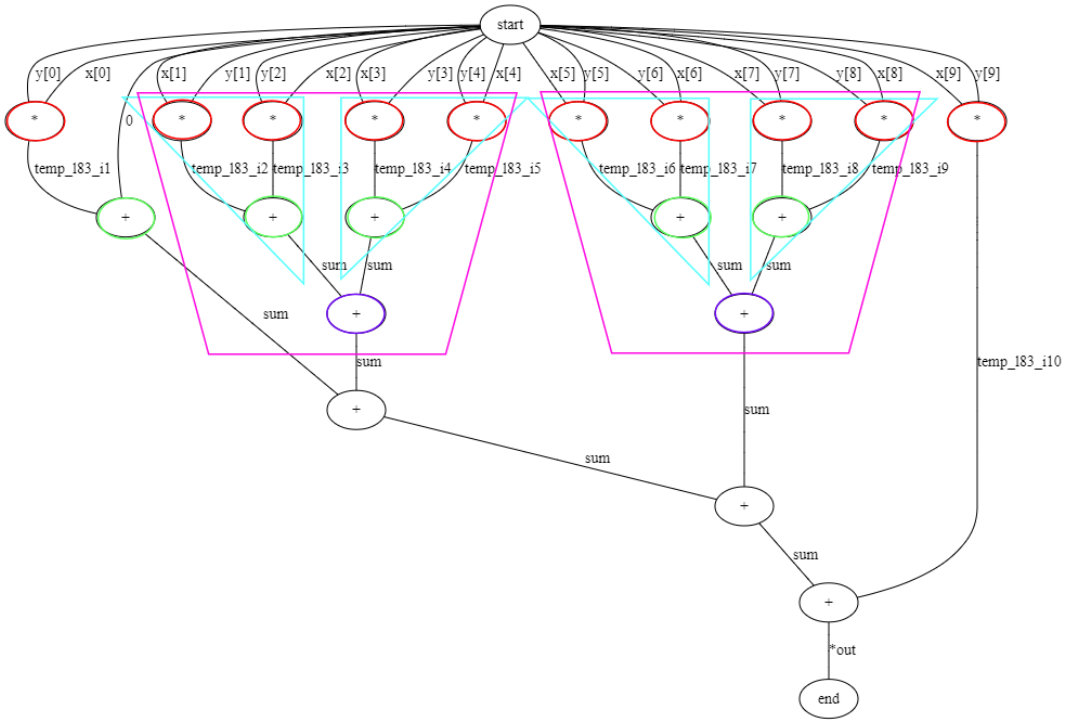
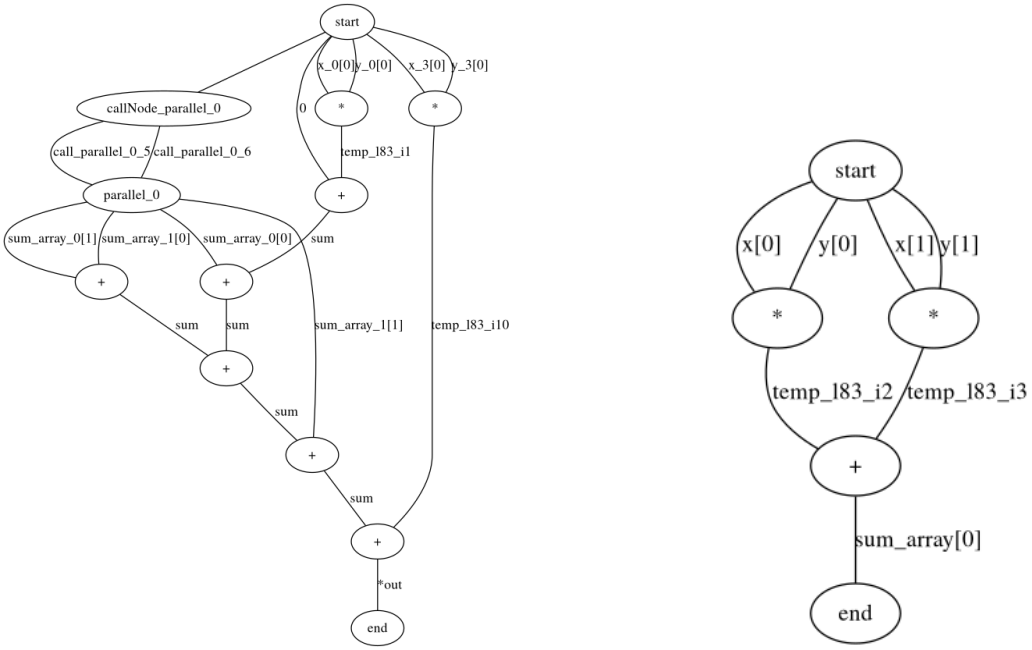



Figure 4.13: Parallel subgraphs colorized for the *Dotprod* graph with $N = 10$.

functions, a call node and a function node are created. Each edge from a call node to a function node represents a new function call. Thus, each edge contains a list with the call arguments to be used. The next step is to connect all the input edges of the parallel subgraphs to the call node. This step is not strictly needed. Another option would be to simply remove all those edges and to add a single edge to the call node with no information, which would generate a less complex graph. However, we have implemented the first option, has it describes best the graph dependencies. Next, the algorithm adds the edges that leave the function node. These are in fact the same edges that are outputs of the parallel subgraphs. Then, the subgraphs list is ordered using the input edges of each subgraph. The next step is to transform scalar inputs and outputs into arrays. Afterwards, the nodes that exist in the parallel subgraphs are removed from the main graph. We proceed to analyze the minimum and maximum accesses on each variable in the parallel subgraphs to compute where each array should be partitioned. Then, we fold all subgraphs into a single subgraph where each edge contains the variable progression along the loop. When folding, we need to update the array accesses on the first three subgraphs according to the partitions defined previously. We use three subgraphs because it allows us to fully determine the type and factors of the progressions. At the moment we support arithmetic and geometric progressions, however the latter needs further testing. Having folded the graph, we add the call edges in the main graph, from the call node to the function node. The call edge syntax is defined in Table 4.3. Lastly, the tool splits the arrays defined as inputs or local variables in the main graph, according to the partitions defined previously, and updates the respective array accesses.




(a) Dotprod main graph after folding.

(b) Dotprod parallel graph folded.

Figure 4.14: Main and parallel graphs generated after the folding stage for the *Dotprod* DFG with $N = 10$.

### 4.4.6 Prologue and Epilogue

All the nodes that were not wrapped into the parallel function need to be wrapped into other functions. This is done to ensure that the code inside the dataflow region is in the canonical form. We start by selecting the nodes to wrap in the prologue. This is done by finding all the "Call" nodes in the main graph and selecting all the nodes in the path upwards until the "Start" node. Then, the nodes selected are wrapped into a new graph that represents the prologue function. Lastly, the algorithm replaces all the nodes selected previously by new "Call" and "Function" nodes. The same process undergoes for the epilogue, but there is a difference in the selection process: any node that is not a "Call", "Function", "Start", or "End" node is selected to be moved into the epilogue. In our *Dotprod* example there are no nodes between the "Start" and the "Call" node. Therefore, only the *Epilogue* graph is created. The final result is illustrated in Figure 4.16.

### 4.4.7 Arithmetic optimizations

All the functions available in the "math.h" library take double as an argument and return double as the result. The arithmetic optimizations algorithm replaces these functions by their respective float

Table 4.3: Call edge attributes.

| **att1** | **att2** | **att3** |
|---|---|---|
| "call" | function name | parameters list |

**Input:** main graph $MG$ and a list of isomorphic subgraphs $IS$
**Output:** main graph $MG$ and folded graph $FG$

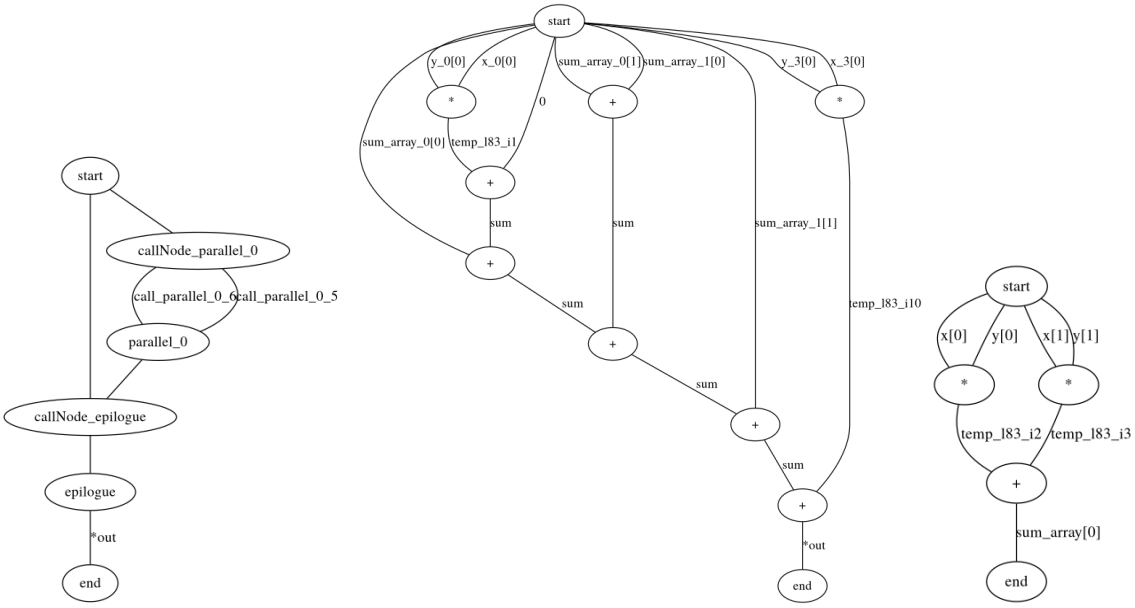
1: CREATECALLNODE($MG$)
2: CREATEFUNCTIONNODE($MG$)
3: ORDERSUBGRAPHS($IS$)
4: INPUTSTOARRAYS($IS, MG$) ▷ Transforms $IS$ inputs into arrays and updates the variables in $MG$ as well.
5: UPDATEOUTPUTS($IS, MG$) ▷ Gives the same label to scalar outputs across subgraphs.
6: OUTPUTSTOARRAYS($IS, MG$) ▷ Transforms $IS$ outputs into arrays and updates the variables in $MG$ as well.
7: REMOVENODESFROMMAINGRAPH($IS, MG$) ▷ Removes the nodes in $IS$ from the main graph.
8: COMPUTEMANUALPARTITIONS($IS.first, IS.last$) ▷ Sets the indexes where each var should be split.
9: $FG \leftarrow$ FOLDSUBGRAPHS($IS$)
10: UPDATEFOLDWIDTH($FG, parallelDegree$)
11: COMPUTEINPUTANDOUTPUTVARS($FG$)
12: ADDCALLEDGES($MG, parallelDegree$) ▷ Adds edges from the Call node to the Function node in $MG$.
13: SPLITARRAYS($MG, parallelDegree$)

Figure 4.15: Fold parallel subgraphs algorithm.

versions. This step appends an "f" to the "mod" attribute of nodes which "mod" attribute matches a predefined list of math functions, e.g. "sqrt" becomes "sqrtf". This algorithm can in the future be extended to, for example, replace multiplications by powers of two by shift operations.

## 4.5 Summary

The framework proposed consists of two main stages, the frontend and the backend. At the frontend, we instrument an ANSI C application and execute it to generate execution traces. These traces are written in the DOT language and are a direct representation of an acyclic DFG. However, our traces differ from the common debug traces because they do not make direct use of any input data. However, if the application contains a loop that executes $x$ times, then the DFG will contain operations that represent the loop body $x$ times. This means that the DFG is only valid for inputs that generate a loop with $x$ iterations. Simple branch conditions need to be manually converted into ternary operators, which are represented in the DFG as "Multiplexer" nodes and require all of its operands to be executed beforehand. Currently, the DFG is written to a file and then processed by the backend. The backend reads the DFG and the configuration file and adds the Start and End nodes. These nodes are useful because we consider that every graph is a direct representation of a code function. Thus, it makes sense that edges leaving the Start node represent inputs of such function, while edges entering the End node represent outputs. Moreover, these nodes provide a starting point for most of the algorithms that follow. Figure 4.4 summarizes the flow of the backend execution. The core transformations focus on the extraction of data-level




(a) Final main graph. (b) Epilogue graph. (c) Parallel graph.

Figure 4.16: Final Main, Epilogue, and Parallel graphs for the *Dotprod* with $N = 10$.

parallelism using isomorphic graph clustering. This kind of parallelism can be coupled with task-level pipelining to achieve significant performance. However, task-level pipelining in HLS needs to follow a set of guidelines which requires aggressive graph restructuring techniques, described in the *Fold Parallel Subgraphs* and *Prologue & Epilogue* algorithms. The following chapter presents the evaluation methodology and results achieved by applying the framework to a set of FPGA benchmarks.

# Chapter 5

# Experimental Results

This chapter outlines the results obtained by the proposed framework when applied to a series of benchmarks. The instrumentation of the benchmarks was generated partially automatically using Clava [10] and the backend processed the graphs and generated C code with directives optimized for Vivado HLS. The results obtained depend on configuration parameters that the user has to provide to the backend. Section 5.1 explains the setup used to obtain the results shown in the following sections. Section 5.2 describes each benchmark, giving examples of real world applications and justifying the need for FPGA acceleration. In addition, it explains the algorithmic structure and highlights the code sections in which we can exploit parallelism. In Section 5.3, we provide detailed metrics from the synthesis process, such as resource usage and execution delay. Section 5.4 evaluates the scalability of the approach in terms of the size of the inputs used in each benchmark. Finally, Section 5.5 is dedicated to evaluating the results and comparing them to the ones reported in state of the art.

## 5.1 Experimental Setup

The benchmarks used are all operation heavy, since they are not control-dominated. The output C code is synthesized using Vivado HLS 2019.2 targeting an Artix™-7 FPGA Xilinx (xc7z020clg484-1, see Table 5.1) . The backend was executed in a computer with a Ryzen 5 3600 and 16GB of RAM. From the synthesis reports, we take for each benchmark the estimated number of LUTs, DSPs, FFs, BRAMs, identified as resource usage. The reports give us also the estimated latency, which is the number of clock cycles necessary to complete the kernel in hardware. The clock period reported is the minimum time in nanoseconds that each cycle takes to execute. The execution delay is calculated by multiplying the latency by the clock period. The area is defined as an average of the percentages of all the resources used:

$$R = \{LUT, DSP, FF, BRAM\} \tag{5.1}$$

$$Area = \frac{1}{4} \sum_{r \in R} \frac{r_{used}}{r_{available}} \tag{5.2}$$

Since resource usage may be as important as execution delay in FPGAs, the objective function, $h(x)$, to minimize is given by:

$$h(x) = Delay(x) \times Area(x) \tag{5.3}$$

, where $x$ is a synthesizable solution. The $h(x)$ values reported are normalized in relation to the $h(x)$ value of the unmodified code. The synthesized solutions are identified by the benchmark name, size, and parallel degree. An undefined parallel degree represents the unmodified code. A parallel degree $p$ of value $p = 0$ represents unfolded code where all instructions are flattened in a single function. A parallel degree $p > 0$ represents code that goes through the data parallelism transformations and contains $p$ pipelined calls to the *parallel* function.

Table 5.1: xc7z020clg484-1 available resources

| BRAM | DSP | FF | LUT |
|---|---|---|---|
| 280 | 220 | 106400 | 53200 |

## 5.2 Benchmarks Description

This section introduces the benchmarks used to evaluate the developed framework. Only 3 benchmarks were used because each one introduced new features that had to be developed into the framework and the input DFG. Table 5.2 summarizes each benchmark. Analyzing table 5.3, it is clear that the restructuring transformations result in higher code complexity with a larger number of lines, functions, and array declarations.

Table 5.2: Benchmarks information.

| Benchmark | Source | Loops | Nested structure | Time complexity | Flow depends on input | If conditions | Brief description |
|---|---|---|---|---|---|---|---|
| SVM | Paper [96] | 2 | Single Nested Loop | O(N_sv * D_sv) | false | true | Support Vector Machine (SVM) kernel |
| Dotprod | DSPLIB [50] | 1 | Single Loop | O(N) | false | false | Takes two vectors and calculates their vector product. The inputs are 16-bit numbers, and the result is a 32-bit number. |
| kNN | In-House | 3 | Outer Loop with 2 inner loops | O(N*(D+K)) | true | true | Assigns to an unclassified data point the classification of the K nearest previously classified data points. |

### 5.2.1 SVM

Support-Vector Machines (SVM) [96] are supervised machine learning models used for data classification. An SVM is trained to classify an input vector into one of two classes. Using a kernel function $K$, it is capable of performing non-linear classifications. The kernel function used in this benchmark is the Radial Basis Function (RBF), which for two vectors $a$ and $b$ is defined as:

$$K(a,b) = \exp(-\gamma\|a-b\|) \tag{5.4}$$

Table 5.3: The complexity of the code generated vs the unmodified versions (first row of each benchmark).

| Benchmark | Parallel Degree | #Lines | #Functions | #Loops | #Arrays | Directives |
|---|---|---|---|---|---|---|
| SVM 1274SV*18D | - | 16 | 1 | 2 | 3 | - |
| | 2 | 1800 | 3 | 1 | 11 | 1 pipeline<br>1 dataflow |
| | 15 | 2460 | 4 | 2 | 50 | 2 pipelines<br>1 dataflow |
| Dotprod N2000 | - | 5 | 1 | 1 | 2 | - |
| | 2 | 420 | 3 | 1 | 10 | 1 pipeline<br>1 dataflow<br>10 array partitions |
| | 8 | 530 | 4 | 2 | 28 | 2 pipelines<br>1 dataflow<br>28 array partitions |
| kNN 8P*128F | - | 65 | 1 | 3 | 5 | - |
| | 0 | 4503 | 1 | 0 | 5 | - |
| | 1 | 917 | 3 | 1 | 6 | 1 pipeline<br>1 dataflow<br>2 array partitions |

The original SVM code is available in Listing A.1.

The SVM kernel used contains a very big outer loop (1274 iterations) and a small inner loop (18 iterations), thus the smaller loop is a good candidate for loop unrolling. Moreover, by allocating the results calculated in the outer loop into a temporary array, we eliminate any dependency between iterations and the values can be added together in the end. Thus, this is a kernel with high instruction level parallelism which can be exploited into data level parallelism. To achieve this kind of parallelism, the backend splits the *sup_vectors*, the *sv_coef* and the temporary array mentioned before into multiple arrays, depending on the parallelism degree set in the configuration file.

The first row in Table 5.3 compares the complexity of the unmodified SVM code with the versions $p = 2$ and $p = 15$ produced by the framework. It is clear that manually transforming the original C code into the any of those versions is almost infeasible.

### 5.2.2 Dot Product

The *dotprod* benchmark [50] consists of a kernel that computes the dot product of two vectors and returns a single scalar number. Since the dot product is an algebraic operation, it is widely used, and thus it is a good candidate for FPGA acceleration. In this benchmark we have used vectors with size $N = 2000$ and $N = 4000$. The dot product can be parallelized by storing the intermediate *sum* values into an array and reducing it in the end, thus eliminating the dependencies between loop iterations. As there is only one loop, the multiplication operations can all be calculated in parallel. However, in practice we are restricted by the FPGA available resources. The benchmark code is available in Listing A.2.

### 5.2.3 kNN

The k-Nearest Neighbours [20] classifier is a machine learning algorithm, which assigns to an unclassified sample data point the classification of the K nearest previously classified data points. The implementation used was developed in-house and uses $K = 3$. This benchmark is used to demonstrate the ability of the framework to deal with ternary operations and with array accesses

that depend on input parameters. The original code is displayed in Listing A.3. For each classified data point it calculates its distance to the unclassified data point and updates the best 3 data points using the distance calculated previously. The execution trace of this kernel shows that there are no dependencies between distances calculations, but the *updateBest* calls need to be executed sequentially. Therefore, this kernel does not present so much ILP as the other kernels tested.

## 5.3 Synthesis Results

This section provides the synthesis results obtained for each benchmark. It reports the speedups and objective function gains compared to the original source codes.

### 5.3.1 SVM

Table 5.4 contains the results of synthesizing the original SVM kernel [96] and for the output of the backend. The output C code uses the dataflow directive to make parallel calls to the kernel while pipelining its outer loop and manually unrolling the inner loop. Listing B.1 shows an example of the code produced. Note that the support vectors and the test vector are already stored in the FPGA RAM. Thus, the speedups achieved do not take into account input delays. An advantage of declaring these variables in the local scope of the top-level function is that Vivado HLS is capable of automatically defining the optimal partitions. However, in a practical scenario we would need to have at least the test vector defined as an argument of the top-level function.

Table 5.4: SVM synthesis results using 1274 support vectors with 18 features each.

| Parallel Degree | Clock (ns) | Latency (cycles) | FF (%) | LUT (%) | BRAM (%) | DSP (%) | Area (%) | Delay (ms) | h(x) | Speedup |
|---|---|---|---|---|---|---|---|---|---|---|
| - | 8.23 | 333826 | 3.3 | 11.6 | 23.6 | 20.5 | 15 | 2.747 | 1.000 | - |
| 1 | 10.25 | 1977 | 11.5 | 25.3 | 1.4 | 10.0 | **12** | 0.020 | 0.006 | 135.6× |
| 2 | 10.25 | 1033 | 12.3 | 33.3 | 1.4 | 16.4 | 16 | 0.011 | 0.004 | 259.4× |
| 4 | 9.92 | 556 | 18.0 | 44.4 | 2.9 | 30.9 | 24 | 0.006 | **0.003** | 498× |
| 8 | 9.72 | 317 | 24.3 | 67.2 | 5.7 | 58.2 | 39 | 0.003 | **0.003** | 891.9× |
| 15 | 9.63 | 205 | 34.6 | 98.1 | 10.7 | 100.0 | 61 | **0.002** | **0.003** | 1392× |

Table 5.4 shows the synthesis report for the SVM benchmark varying the parallelism degree $p$. For this benchmark we are limited by the number of LUTs and DSPs available. Version $p = 15$ reached a speedup of 1392× with 15 calls to the *parallel* function. Versions $p = 4$, $p = 8$, and $p = 15$ minimized the objective function $h(x)$, meaning that they achieved the best balance between speedup and area used. Figure 5.1 shows the speedup evolution as the parallelism degree is increased.

### 5.3.2 Dot Product

Table 5.5 shows the synthesis results for the *Dotprod* benchmark with $N = 2000$. The code generated by the framework uses the dataflow directive to pipeline the parallel calls to the *Dotprod* kernel. One may notice that the latency is not inversely proportional to the number of parallel

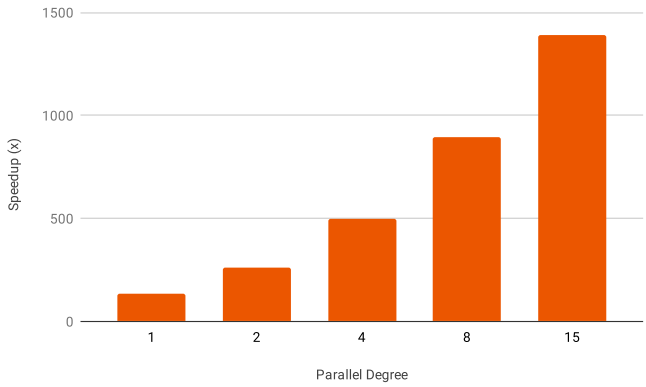



Figure 5.1: Speedups for the SVM benchmark with 1274 support vectors and 18 features.

calls, and that is expected because only the multiplications are executed in parallel. The latency corresponds to the cycles used for kernel execution plus the cycles used in the epilogue, which reduces the values calculated in the kernel into a single scalar. For $N = 2000$ the version with $p = 8$ has a speedup of 122.8× compared to the unmodified C code. With respect to $h(x)$, the unmodified source code performs the best. The latter is explained by the fact that the original C code uses a low number of FPGA resources (0.2%).

Table 5.5: Synthesis results for the *Dotprod* benchmark, $N = 2000$

| Parallel Degree | Clock (ns) | Latency (cycles) | FF (%) | LUT (%) | BRAM (%) | DSP (%) | Area (%) | Delay (ns) | h(x) | Speedup |
|---|---|---|---|---|---|---|---|---|---|---|
| - | 6.4 | 6001 | 0.1 | 0.1 | 0.0 | 0.5 | 0.2 | 38286.4 | **1.000** | - |
| 1 | 10.8 | 165 | 3.6 | 10.1 | 1.4 | 14.5 | 7.4 | 1773.9 | 2.043 | 21.6× |
| 2 | 10.8 | 86 | 4.8 | 15.0 | 0.0 | 21.8 | 10.4 | 924.6 | 1.495 | 41.4× |
| 3 | 10.8 | 61 | 5.8 | 16.5 | 0.0 | 29.1 | 12.8 | 655.8 | 1.308 | 58.4× |
| 4 | 10.8 | 48 | 6.7 | 18.0 | 0.0 | 36.4 | 15.3 | 516.0 | 1.224 | 74.2× |
| 5 | 10.8 | 40 | 7.7 | 19.2 | 0.0 | 43.6 | 17.6 | 430.0 | 1.178 | 89.0× |
| 6 | 10.8 | 36 | 8.5 | 20.1 | 0.0 | 50.9 | 19.9 | 387.0 | 1.195 | 98.9× |
| 7 | 10.8 | 32 | 9.7 | 21.2 | 0.0 | 58.2 | 22.3 | 344.0 | 1.190 | 111.3× |
| 8 | 10.8 | 29 | 10.5 | 22.2 | 0.0 | 65.5 | 24.5 | **311.8** | 1.188 | 122.8× |

Table 5.6 contains the synthesis results of the code produced by the framework with $N = 4000$. Only the unmodified source code and $p = 1$ would actually be possible to implement in the selected FPGA, because versions with $p > 1$ overuse DSPs. Nonetheless, it is interesting to notice that the number of cycles is actually lower than the corresponding version in table 5.5. This happens because the framework detects larger parallel node clusters, thus it is able of performing more computation in each cycle, but at the cost of using more resources.

As most of the rows in Table 5.6 represent RTL that does not fit the FPGA, we decided to assess the code generated by the framework when limiting the maximum number of nodes in each parallel cluster. Table 5.7 shows the synthesis results for the *Dotprod* benchmark with $N = 4000$ and the isomorphic subgraphs size limited to 33 nodes. Figure 5.2 shows the speedups evolution with respect to the parallelism degree. We limited the amount of nodes in each subgraph because otherwise the hardware designs would not fit in the FPGA. Limiting the nodes in each isomorphic

Table 5.6: Synthesis results for the *Dotprod* benchmark, $N = 4000$. The results with resources above 100% are merely indicative and are not implementable in the target FPGA.

| Parallel Degree | Clock (ns) | Latency (cycles) | FF (%) | LUT (%) | BRAM (%) | DSP (%) | Area (%) | Delay (ns) | h(x) | Speedup |
|---|---|---|---|---|---|---|---|---|---|---|
| - | 6.4 | 12001 | 0.1 | 0.1 | 0 | 0.5 | 0.2 | 76566.4 | **1.000** | - |
| 1 | 10.8 | 86 | 9.9 | 25.0 | 0 | 72.7 | 26.9 | **924.6** | 1.933 | 82.8× |
| 2 | 10.8 | 48 | 13.5 | 44.8 | 0 | **101.8** | 40.0 | 516.0 | 1.605 | 148.4× |
| 3 | 10.8 | 36 | 16.1 | 47.6 | 0 | **130.9** | 48.6 | 387.0 | 1.462 | 197.8× |

subgraph is usually an effective method to reduce the total area used, because it trades computation inside the loop by more cycles. Again, the code with $p = 8$ achieves the highest speedup, 145.3×, while the unmodified code minimizes $h(x)$. Limiting the number of nodes in each subgraph to 33 resulted in finding the same isomorphic subgraph for N=4000 and for N=2000. Therefore, the *parallel* function for both input sizes looks the same, except for the number of iterations of the loop, which is 249 for $N = 4000$ and 124 for $N = 2000$. The *parallel* function in $N = 4000$ takes twice the number of cycles compared to $N = 2000$. However, the epilogues takes roughly the same time to execute. This explains why the delays observed in $N = 4000$ are always less than twice the delays observed in $N = 2000$ for the same value of $p$, $p > 0$.

Table 5.7: Synthesis results for the *Dotprod* benchmark, $N = 4000$ and $maxNodesPerSubgraph = 33$.

| Parallel Degree | Clock (ns) | Latency (cycles) | FF (%) | LUT (%) | BRAM (%) | DSP (%) | Area (%) | Delay (ns) | h(x) | Speedup |
|---|---|---|---|---|---|---|---|---|---|---|
| - | 6.4 | 12001.0 | 0.1 | 0.1 | 0.0 | 0.5 | 0.2 | 76566.4 | **1.000** | - |
| 1 | 10.8 | 322.0 | 4.2 | 13.6 | 1.4 | 14.5 | 8.4 | 3461.8 | 2.268 | 22.1× |
| 2 | 10.8 | 167.0 | 5.8 | 18.5 | 2.9 | 21.8 | 12.3 | 1795.4 | 1.709 | 42.6× |
| 3 | 10.8 | 114.0 | 7.2 | 20.1 | 4.3 | 29.1 | 15.2 | 1225.6 | 1.443 | 62.5× |
| 4 | 10.8 | 88.0 | 7.6 | 21.6 | 2.1 | 36.4 | 16.9 | 946.1 | 1.244 | 80.9× |
| 5 | 10.8 | 72.0 | 8.9 | 23.6 | 0.0 | 43.6 | 19.0 | 774.1 | 1.145 | 98.9× |
| 6 | 10.8 | 62.0 | 9.9 | 25.2 | 0.0 | 50.9 | 21.5 | 666.6 | 1.114 | 114.9× |
| 7 | 10.8 | 54.0 | 11.1 | 27.4 | 0.0 | 58.2 | 24.2 | 580.6 | 1.090 | 131.9× |
| 8 | 10.8 | 49.0 | 11.5 | 26.1 | 0.0 | 65.5 | 25.8 | **526.8** | 1.054 | 145.3× |

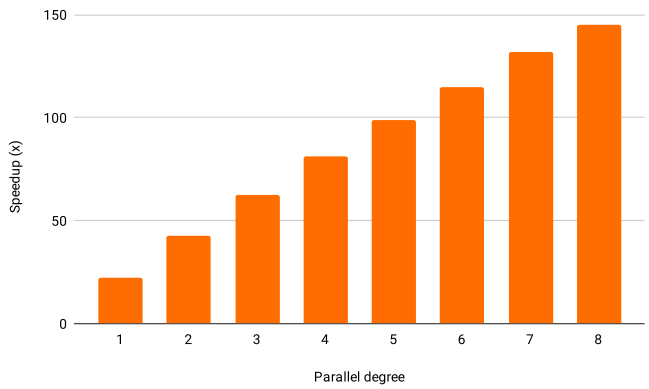



Figure 5.2: Speedups for the *Dotprod* benchmark, $N = 4000$ and $maxNodesPerSubgraph = 33$.

### 5.3.3 kNN

When the kNN DFG was processed by the framework and the output code synthesised, Vivado HLS reported in most cases an overuse of LUTs. Thus, for this benchmark we had to turn the *saveEnergy* option on in the configuration file (only when the *fold* option was on), which generates sub-optimal array partitions that balance area with performance. We evaluated this benchmark for 3 different input sizes, all with 8 data points, and with 32, 64 and 128 features per data point. In the kNN version used, only the distances calculation can be executed in parallel. Thus, when folding and restructuring the code to use the dataflow directive, the framework detects the distances calculation as parallelizable and wraps that section into a function. The remainder of the code goes into the epilogue function. Listing B.3 contains an example of the code produced by the framework.

Tables 5.8, 5.9, and 5.10 show the synthesis results of the kNN benchmark for multiple input sizes, considering the types of the operations and of the arrays xFeatures and knownFeatures to be "double". For all of the inputs written by the backend, the *arithmetic* and *parallelizeSums* optimization was set to "true", while the *pruneLocalArrays* configuration was set to "false".

For the benchmark with 8 data points with 32 features each, the best valid speedup is accomplished by the $p = 4$ input, which runs $14\times$ faster than the unmodified code. However, the objective function $h(x)$ is minimized by the *No Fold* input, $p = 0$. In this benchmark, the input with $p = 8$ overused LUTs.

Table 5.8: Synthesis results of the code produced by the framework for the kNN benchmark with 8 data points and 32 features. The results with resources above 100% are merely indicative and are not implementable in the target FPGA.

| Parallel Degree | Clock (ns) | Latency (cycles) | FF (%) | LUT (%) | BRAM (%) | DSP (%) | Area (%) | Delay (ns) | h(x) | Speedup |
|---|---|---|---|---|---|---|---|---|---|---|
| - | 8.6 | 4954 | 3.6 | 12.4 | 0 | 6.4 | 5.6 | 42718 | 1 | - |
| 0 | 10.7 | 291 | 7.8 | 30.6 | 0 | 7.3 | 11.4 | 3123 | **0.15** | 13.68× |
| 1 | 10.6 | 311 | 22.3 | 65.0 | 0.4 | 40.0 | 31.9 | 3282 | 0.44 | 13.01× |
| 2 | 9.4 | 313 | 32.8 | 86.5 | 0.4 | 43.6 | 40.8 | **2943** | 0.50 | 14.51× |
| 4 | 9.3 | 321 | 37.1 | 88.6 | 0.4 | 29.1 | 38.8 | 2983 | 0.48 | 14.32× |
| 8 | 9.3 | 171 | 52.2 | **153.2** | 0.4 | 50.9 | 64.2 | 1589 | 0.43 | 26.88× |

To understand how the resource usage scales with the *saveEnergy* option activated, we proceed to analyze the results for 8 data points and 64 features from Table 5.9. This time, both the $p = 4$ and $p = 8$ versions overuse LUTs. Thus, the best valid speedup is achieved by the code with $p = 2$, which was $23.47\times$ faster compared to the unmodified version. $h(x)$ is minimized by the unfolded code, $p = 0$.

For the the 8 data points and 128 features benchmark (see Table 5.10), the versions with $p = 4$ and $p = 8$ overused resources as expected. Thus, the best valid speedup is achieved by $p = 1$, which was $20.43\times$ faster than the unmodified code. The objective function is again minimized by the unfolded code, $p = 0$.

Examining the previous results, we conclude that the inputs which use the dataflow directive always achieve the best speedups. However, when we consider the area used to be as important as

Table 5.9: Synthesis results of the code produced by the framework for the kNN benchmark with 8 data points and 64 features. The results with resources above 100% are merely indicative and are not implementable in the target FPGA.

| Parallel Degree | Clock (ns) | Latency (cycles) | FF (%) | LUT (%) | BRAM (%) | DSP (%) | Area (%) | Delay (ns) | h(x) | Speedup |
|---|---|---|---|---|---|---|---|---|---|---|
| - | 8.6 | 9562 | 3.6 | 12.4 | 0 | 6.4 | 5.6 | 82453 | 1 | - |
| 0 | 11.1 | 451 | 8.0 | 34.3 | 0 | 7.3 | 12.4 | 5006 | **0.13** | 16.47× |
| 1 | 10.6 | 359 | 22.3 | 53.6 | 0.4 | 29.1 | 26.3 | 3789 | 0.22 | 21.76 |
| 2 | 9.3 | 378 | 32.0 | 82.3 | 0.4 | 29.1 | 35.9 | **3513** | 0.27 | 23.47 |
| 4 | 9.3 | 381 | 45.9 | **115.4** | 0.4 | 29.1 | 47.7 | 3562 | 0.37 | 23.15 |
| 8 | 9.5 | 171 | 76.1 | **222.0** | 0.4 | 76.4 | 93.7 | 1632 | 0.33 | 50.53 |

the delay, then the *No Fold* version performs best. Observing the code generated by the framework, there are ternary operations which have the same argument on the true and false sides. This could be optimized by searching through all multiplexer nodes and reducing these operations into simple assignments.

Table 5.11 holds the synthesis results considering the xFeatures and knowsFeatures arrays as floats as removing the the call to the "sqrt" function in the calculation of the Euclidean distances. We only report the results for an input size of 8 points and 128 features as this is the most meaningful of the three input sizes considered previously. The use of the "sqrt" function is not expected to have a significant impact, due to the low number of classified data points used. To study the impact of changing the data size from doubles to floats, we can compare the best results from Table 5.10 and Table 5.11. We conclude that there is a 60.6% delay reduction for $p = 1$ and that the function $h(x)$ is now minimized by $p = 1$ instead of $p = 0$. Figure 5.3 summarizes the speedups achieved across the different input sizes and data types.

One weak point of the backend is that it does not implement yet detection and folding of sequential patterns. As the *updateBest* instructions are sequential in the DFG, they are moved into the epilogue. Thus, the epilogue grows linearly with the the number of classified data points. Moreover, as we increase the number of classified data points, the percentage of parallel code decreases. This explains why the results presented for this benchmark have a low number of data points compared to the number of features.

Nevertheless, to demonstrate the potential of the framework, we experimented processing the DFG with more realistic sizes while manually rewriting the epilogue. The results are shown in

Table 5.10: Synthesis results of the code produced by the framework for the kNN benchmark with 8 data points and 128 features. The results with resources above 100% are merely indicative and are not implementable in the target FPGA.

| Parallel Degree | Clock (ns) | Latency (cycles) | FF (%) | LUT (%) | BRAM (%) | DSP (%) | Area (%) | Delay (ns) | h(x) | Speedup |
|---|---|---|---|---|---|---|---|---|---|---|
| - | 8.6 | 18778 | 3.6 | 12.4 | 0 | 6.4 | 5.6 | 161923 | 1 | - |
| 0 | 11.3 | 771 | 8.4 | 39.6 | 0 | 7.3 | 13.8 | 8678 | **0.13** | 18.66× |
| 1 | 10.6 | 751 | 22.3 | 52.6 | 0.4 | 21.8 | 24.3 | **7926** | 0.21 | 20.43× |
| 2 | 9.4 | 888 | 41.8 | 98.4 | 0.4 | 14.5 | 38.8 | 8377 | 0.36 | 19.33× |
| 4 | 9.4 | 760 | 81.4 | **182.7** | 0.4 | 29.1 | 73.4 | 7169 | 0.58 | 22.59× |
| 8 | 9.6 | 436 | **125.1** | **351.7** | 0.4 | 83.6 | **140.2** | 4182 | 0.65 | 38.72× |

Table 5.11: Synthesis results of the code produced by the framework for the kNN benchmark with 8 data points, 128 features. Float input data and calculations. Removed the "sqrt" from the calculation of the Euclidean distances.

| Parallel Degree | Clock (ns) | Latency (cycles) | FF (%) | LUT (%) | BRAM (%) | DSP (%) | Area (%) | Delay (ns) | h(x) | Speedup |
|---|---|---|---|---|---|---|---|---|---|---|
| - | 8.40 | 16498 | 1.1 | 4.6 | 0.0 | 2.3 | 2.0 | 138500.7 | 1 | - |
| 0 | 10.28 | 582 | 16.0 | 45.9 | 0.0 | 16.8 | 19.7 | 5983.5 | 0.43 | 23.15× |
| 1 | 9.58 | 326 | 10.1 | 31.9 | 0.4 | 19.1 | 15.4 | **3122.4** | **0.17** | 44.36× |
| 2 | 8.75 | 450 | 17.2 | 61.6 | 0.4 | 12.7 | 23.0 | 3937.5 | 0.33 | 35.17× |

Table 5.12. All results used float data types and had no call to the *sqrt* function. Column "II" represents the initiation interval of the loop in the parallel function. To achieve these results, we manually increased the II until a valid implementation was obtained. To get the most out of the dataflow pipelining, we extracted data-level parallelism out of the *updateBest* task and the results were merged in the end. An example of the code produced is shown in Listing B.4. The best speedups range from 97.89× to 164.28×, showing that the approach achieves consistent results for all the input sizes considered. The biggest concern in terms of area is with the number of estimated LUTs, which reaches values close to 100%.

Table 5.12: Synthesis results of the code produced by the framework for the kNN benchmark with the epilogue manually optimized. Only the unmodified and best results are shown for each input size.

| Input Size | Parallel Degree | II | Clock (ns) | Latency (cycles) | FF (%) | LUT (%) | BRAM (%) | DSP (%) | Area (%) | Delay (ms) | h(x) | Speedup |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1000p,32f | - | - | 8.40 | 535010 | 1.5 | 5.9 | 0 | 2.3 | 2.4 | 4.4914 | 1 | - |
| | 4 | 8 | 8.64 | 4299 | 37.2 | 85.0 | 1.4 | 50.9 | 43.7 | 0.0371 | **0.15** | 120.99× |
| 1000p, 64f | - | - | 8.40 | 1047010 | 1.5 | 5.9 | 0 | 2.3 | 2.4 | 8.7896 | 1 | - |
| | 4 | 16 | 8.64 | 6196 | 34.2 | 99.4 | 1.4 | 50.9 | 46.5 | 0.0535 | **0.12** | 164.28× |
| 1000p, 128f | - | - | 8.40 | 2071010 | 1.5 | 5.9 | 0 | 2.3 | 2.4 | 17.3861 | 1 | - |
| | 2 | 16 | 8.74 | 12567 | 45.8 | 99.5 | 0.7 | 50.9 | 49.2 | 0.1098 | **0.13** | 158.27× |
| 10000p, 16f | - | - | 8.40 | 2790010 | 1.5 | 5.9 | 0 | 2.3 | 2.4 | 23.4221 | 1 | - |
| | 4 | 3 | 8.64 | 27710 | 29.1 | 84.4 | 11.4 | 72.7 | 49.4 | 0.2393 | **0.21** | 97.89× |
| 10000p, 32f | - | - | 8.40 | 5350010 | 1.5 | 5.9 | 0 | 2.3 | 2.4 | 44.9133 | 1 | - |
| | 4 | 6 | 8.64 | 35216 | 39.4 | 97.6 | 11.4 | 72.7 | 55.3 | 0.3041 | **0.15** | 147.70× |

### 5.3.4 Summary

Although none of the 3 benchmarks tested are originally written as producer-consumer algorithms, the developed framework is capable of restructuring the code to use the *dataflow* directive and extract better results than the original versions, both in terms of speedup and $Area \times Delay$. The SVM kernel proved to be an ideal candidate for this transformation, achieving a 1392× speedup with great efficiency $h(x) = 0.003$. The *dotprod* benchmark is used to evaluate the power of limiting the number of nodes in each parallel cluster to produce results that are more energy-efficient, as area is closely related to the energy used. Nonetheless, in this benchmark we have speedups of up to 122.8×. The kNN benchmark is the most complex one, as it has conditional operations, array accesses than depend on inputs, and a lower amount of ILP. It was used to demonstrate that this approach is not limited to simple kernels. Although the results do not show speedups as high as the other benchmarks, we still see speedups of up to 44.36× compared to the

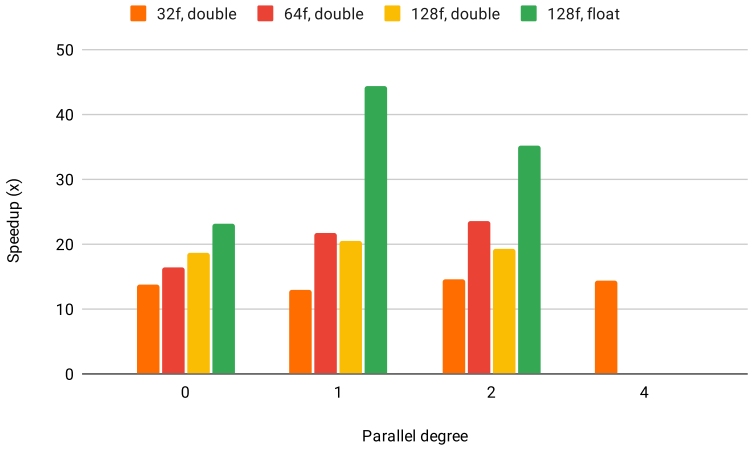



Figure 5.3: Speedups for the KNN benchmark with 8 data points and multiple number of features.

unmodified source code. From the results that we have, it seems that the speedups remain fairly constant as the input sizes increase, but more data is needed to make that conclusion.

## 5.4 Backend Execution Time & Scalability

This section evaluates the scalability of the backend in relation to the size of the input DFG and provides profiling data to better understand the execution bottlenecks. When dealing with large DFGs, the execution time of the backend depends largely on some user provided configurations. The options that affect the execution time the most are the following: *fold*, which tells the backend to search for code that can be executed in parallel and to restructure it to use the dataflow directive; *minFoldLevels*, which sets the minimum number of levels of each parallel cluster; *maxFoldLevels*, which sets the maximum number of levels of each parallel cluster. The kNN benchmark has the less amount of ILP and the number of levels of the DFG increase directly proportional to the dimension of each data point. It happens that the algorithm that finds all combinations of subgraphs in the DFG has an average $\mathcal{O}((l^2 - l) \cdot avg(n_l))$ time complexity, where $l$ is the number of levels in the graph and $n_l$ is the number of nodes per level. Therefore, the kNN benchmark is the one which execution time scales more poorly of the three. To gather the execution times of the kNN benchmark, each test was run 3 times and the times were averaged out. Table 5.13 contains the results.

Figure 5.4 traces the evolution of the execution time with the number of features of each data point. Both axes use a logarithmic scale and the two variables seem to follow a linear relation. This is only possible thanks to the user options *minFoldLevels* and *maxFoldLevels*, which allowed to limit the search space of the parallel clusters. The values for these variables were chosen so that $maxFoldLevels - minFoldLevels <= 50$. Without these configurations, the execution times would scale quadratically, making it unfeasible to execute the backend tool. To estimate how many

levels the folded graph should have for large input DFGs, the user can execute the tool for smaller inputs and extract the levels from the execution logs.

Tables 5.14, 5.15, and 5.16 show the results of profiling each benchmark using the Java Flight Recorder profiler. All of the results shown represent an average of 3 executions. Only the algorithms which used significant CPU time are shown. The "Levels Search Range" column is defined as $maxFoldLevels - minFoldLevels + 1$, e.g., if it is equal to 1 for the SVM benchmark, it means that the "AllSubgraphs" algorithm will only search for subgraphs that have exactly 24 levels because that is the number of levels of the best parallel cluster. Using an optimal search range, we were able to reduce the execution times by 48%, 19%, and 87% for the SVM, dotprod, and kNN benchmarks respectively. The dotprod benchmark execution times are dominated by the "Leveling" algorithm, which happens because the input DFG has 4002 levels. For comparison, the SVM and kNN input DFGs have 1299 and 242 levels respectively. Both the SVM and kNN benchmarks have poor execution times when the search range is not defined. This is expected, because searching for all the possible subgraphs in a graph with 243 levels, as in the kNN, results in a huge number of possible subgraphs. The "FoldParallelSubgraphs" algorithm is the second one to use more CPU time in most of the results, so in the future one may take a better look at it to gain some time.

Table 5.13: Backend execution times for the kNN benchmark using 8 data points and varying the number of features.

| Features | Nodes | Edges | Levels | Execution Time 1 (sec) | Execution Time 2 (sec) | Execution Time 3 (sec) | Average Execution Time (sec) |
|---|---|---|---|---|---|---|---|
| 32 | 1833 | 2556 | 146 | 0.643 | 0.579 | 0.599 | 0.607 |
| 64 | 3145 | 4380 | 178 | 0.855 | 0.824 | 0.813 | 0.831 |
| 128 | 5769 | 8028 | 242 | 1.449 | 1.480 | 1.540 | 1.490 |
| 256 | 11017 | 15324 | 370 | 3.610 | 4.015 | 4.027 | 3.884 |
| 512 | 21513 | 29916 | 626 | 6.051 | 5.844 | 5.971 | 5.955 |
| 1024 | 42505 | 59100 | 1138 | 25.821 | 25.756 | 26.574 | 26.050 |
| 2048 | 84489 | 117468 | 2162 | 43.585 | 46.635 | 43.032 | 44.417 |

Table 5.14: Profiling results for the SVM benchmark. Levels in the best parallel cluster: 24. Levels in graph before executing the AllSubgraphs algorithm: 38. Input size: 1274 support vectors with 18 features each.

| Levels Search Range | AllSubgraphs (%) | FoldParallelSubgraphs (%) | FileParser (%) | Leveling (%) | Pruning (%) | Others (%) | Execution time (sec) |
|---|---|---|---|---|---|---|---|
| No range defined | 69.5 | 10.1 | 4.8 | 5.9 | 4.6 | 5.1 | 13.014 |
| 10 | 47.8 | 15.6 | 9.2 | 9.9 | 8.5 | 9.0 | 8.728 |
| 1 | 11.4 | 27.1 | 15.6 | 16.4 | 14.6 | 14.9 | 6.745 |

In order to improve backend scalability, the pruning algorithm was improved from the one proposed by Ferreira et al. [32] [31]. The efficiency of this algorithm is of great importance because it decreases the execution time of all the algorithms that follow. Table 5.17 shows the pruning efficiency that was evaluated for each benchmark. We achieved about 60% nodes reduction and

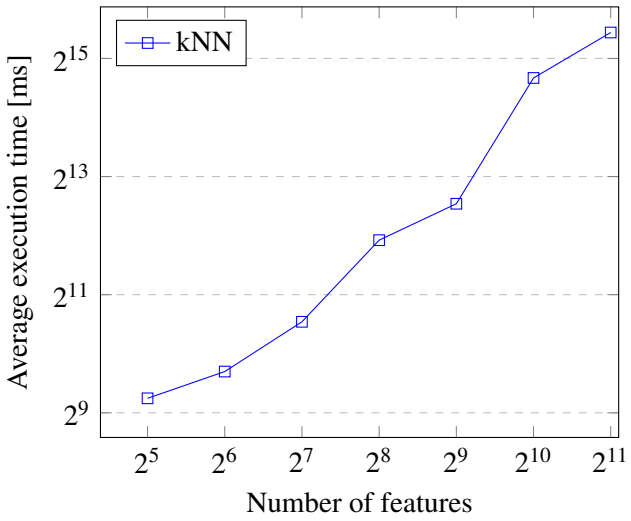



Figure 5.4: Backend execution time for the kNN benchmark using 8 data points.

Table 5.15: Profiling results for the dotprod benchmark. Levels in the best parallel cluster: 6. Levels in graph before executing the AllSubgraphs algorithm: 17. Input size: N=4000.

| **Levels Search Range** | **AllSubgraphs (%)** | **FoldParallelSubgraphs (%)** | **FileParser (%)** | **Leveling (%)** | **Pruning (%)** | **Others (%)** | **Execution time (sec)** |
|---|---|---|---|---|---|---|---|
| No range defined | 14.6 | 5.1 | 4.7 | 68.1 | 0.6 | 6.9 | 4.71 |
| 10 | 10.9 | 5.4 | 4.9 | 72.7 | 0 | 6.1 | 4.298 |
| 1 | 2.5 | 5.7 | 4.5 | 78.4 | 0.6 | 8.2 | 3.809 |

Table 5.16: Profiling results for the kNN benchmark. Levels in the best parallel cluster: 70. Levels in graph before executing the AllSubgraphs algorithm: 243. Input size: 8 data points with 128 features each.

| **Levels Search Range** | **AllSubgraphs (%)** | **FoldParallelSubgraphs (%)** | **FileParser (%)** | **Leveling (%)** | **Pruning (%)** | **Others (%)** | **Execution time (sec)** |
|---|---|---|---|---|---|---|---|
| No range defined | 96.4 | 1.6 | 0.5 | 0.5 | 0.5 | 0.5 | 6.120 |
| 30 | 89.3 | 6.1 | 0.5 | 2.3 | 0.5 | 1.3 | 1.846 |
| 1 | 37.1 | 19.9 | 17.4 | 5.5 | 2.8 | 17.2 | 0.783 |

43% to 50% edges reduction, which is a significant improvement over the 42% node reduction and 24% edge reduction of the previous work.

Table 5.17: Pruning data for each DFG.

| Benchmark | Dataset Size | Nodes before pruning | Edges before pruning | Nodes after pruning | Edges after pruning | Nodes Reduction (%) | Edges Reduction (%) | Levels |
|---|---|---|---|---|---|---|---|---|
| SVM | N_sv = 1274 D_sv = 18 | 126158 | 175847 | 49691 | 99380 | 60.6 | 43.5 | 1299 |
| Dotprod | N=2000 | 12005 | 16004 | 4002 | 8001 | 66.7 | 50.0 | 2002 |
| Dotprod | N=4000 | 24005 | 32004 | 8002 | 16001 | 66.7 | 50.0 | 4002 |
| kNN | N=8, D=32 | 1833 | 2556 | 748 | 1471 | 59.2 | 42.4 | 146 |
| kNN | N=8, D=64 | 3145 | 4380 | 1260 | 2495 | 59.9 | 43.0 | 178 |
| kNN | N=8, D=128 | 5769 | 8028 | 2284 | 4543 | 60.4 | 43.4 | 242 |
| kNN | N=8, D=256 | 11017 | 15324 | 4332 | 8639 | 60.7 | 43.6 | 370 |
| kNN | N=8, D=512 | 21513 | 29916 | 8428 | 16831 | 60.8 | 43.7 | 626 |
| kNN | N=8, D=1024 | 42505 | 59100 | 16620 | 33215 | 60.9 | 43.8 | 1138 |
| kNN | N=8, D=2048 | 84489 | 117468 | 33004 | 65983 | 60.9 | 43.8 | 2162 |

The *Leveling* (see Section 4.7) algorithm is used multiple times depending on the configurations provided. Thus, it is important that it executes as efficiently as possible. For this reason, we developed a new algorithm which is capable of leveling the whole graph while processing each node only once. This algorithm attributes a level to each node and returns a *levelGraph* which lists all nodes that exist in each level. Figure 5.5 shows an execution comparison between the previous algorithm [32], [31] and the new one for a similar SVM graph.

```
1  Iter 0: {Start}
2  Iter 1: {const1, test_vector[0]_0, ..., b_0 } #21
3  Iter 2: {norma_1, op10, op19, op28, ..., op37 } #22
4  Iter 3: {op16, op25, op34, op7, op8, ..., op30} #22
5  Iter 4: {op16, op25, op34, op7, op8, ..., op6} #18
6  Iter 5: {op16, op25, op34, op7, op8, ..., op6} #14
7  Iter 6: {op16, op25, op34, op7, op8, ..., op37} #9
8  Iter 7: {op8, op9, op17, op26, op35, op37}
9  Iter 8: {op9, op37, op18, op27, op36}
10 Iter 9: {op37, op18, op27, op36}
11 Iter 10: {op37, op27, op36}
12 Iter 11: {op37, op36}
13 Iter 12: {op37}
14 Iter 13: {*sum_5}
15 Iter 14: {End}
```

Listing 5.1: Previous Leveling algorithm

```
1  Iter 0: {Start}
2  Iter 1: {op3, op10, op17, op1, op8, op15, op22, op24}
3  Iter 2: {op16, op23, op2, op9}
4  Iter 3: {op11, op25, op18, op4}
5  Iter 4: {op26, op12, op19, op5}
6  Iter 5: {op13, op20, op27, op6}
7  Iter 6: {op21, op7}
8  Iter 7: {op14}
9  Iter 8: {op28}
10 Iter 9: {op29}
11 Iter 10: {op30}
12 Iter 11: {mux_1}
13 Iter 12: {End}
```

Listing 5.2: New Leveling algorithm

Figure 5.5: Execution comparison of the *Leveling* algorithm for a similar SVM DFG.

## 5.5 State of the Art Comparison

This section compares the synthesis results with those reported by state-of-the-art approaches.

### 5.5.1 SVM

When comparing the version with 15 function calls with the *framework-08* version published in [30] (see Table 5.18), there is a 15.6× speedup in terms of latency cycles, and 13.7× in terms of execution delay. *framework-08* used a very similar approach, taking an execution trace as input, but did not explore the use of the dataflow directive. Notice that the numbers reported for *framework-08* were calculated using 1248 support vectors instead of 1274, but we consider the difference to be low enough to allow for a direct comparison.

Table 5.18: SVM synthesis report published in [30]

| Framework | Clock (ns) | Latency (cycles) | FF (%) | LUT (%) | DSP (%) | BRAM (%) | Area used (%) | Delay (ns) | Speedup |
|---|---|---|---|---|---|---|---|---|---|
| Framework-08 | 8.4 | 3208 | 12 | 27 | 41 | 0 | 20 | 26947 | 14.99× |

Tsoutsouras et al. [96] refer a latency gain of 98.78% compared to Vivado HLS default optimized solution, using the same target FPGA and same input size. Our SVM implementation that makes 15 parallel calls to the SVM kernel achieves 99.93% latency gains in respect to the original source code. One might notice that we report different latency for the code with no optimizations (333826 vs 412783), which may be explained by the different Vivado HLS versions used.

Santos et al. [83] used their framework to synthesise the SVM code with Vivado HLS, using the same FPGA but 1248 support vectors instead of 1274. As they have got different latency results for the original source code, it is not fair to compare the latencies directly, but only the speedups achieved. They got a speedup of 24.85× compared to the original source code, while our is of 1392×, making our speedup 56× higher.

Recently, Afifi et al. [2] published a review on 44 papers about FPGA-based SVM classifiers. However, it is not easy to find a meaningful comparison with other implementations, as there is no standard for the sizes of the inputs, target clock, or FPGA to use.

### 5.5.2 Dot Product

Ferreira et al. [30] studied the performance of their approach for the *dotprod* benchmark with $N = 2000$, getting the same results as us in the synthesis of the original source code. They report a 16.8× speedup while we are able to achieve a 122.8× speedup by making 8 parallel calls to the kernel. Such a speedup difference happens because we are able to extract ILP into data-level parallelism.

### 5.5.3 kNN

Santos et al. [83] developed a tool which searches and inserts directives using heuristics. In addition, the tool is capable of providing compile-time optimized mathematical functions for HLS. For the kNN benchmark with an input size of 8 data points and 128 features, the tool adds complete array partitioning to the "xFeatures", "knownFeatures", and "knownClasses" arrays. In addition, it pipelines the outermoost loop which fully unrolls the inner loop. Thus, the major differences

between our approaches are in the partition factors used and in the use of the dataflow directive. Table 5.19 shows the synthesis report obtained by the tool developed by Santos et al. [83], achieving a speedup of 14.62× compared to the unmodified code. Our results for the same input size and data types are available in Table 5.11, where we achieve a speedup of 44.36×.

Table 5.19: kNN synthesis report for the code output by the tool developed by Santos et al. [83]. Input size: 8 data points, 128 features. "xFeatures" and "knownFeatures" with float types.

| **Clock (ns)** | **Latency (cycles)** | **FF (%)** | **LUT (%)** | **BRAM (%)** | **DSP (%)** | **Area Used (%)** | **Delay (ns)** | **Speedup** |
|---|---|---|---|---|---|---|---|---|
| 8.46 | 1120 | 13.1 | 24.6 | 0 | 3.6 | 10.3 | 9472 | 14.62× |

This is a benchmark which was not yet possible to execute using the similar approach developed by Ferreira et al. [30], which layed the ground work for this thesis.

## 5.6 Summary

This chapter presented the evaluation of the framework using the *SVM*, *dotprod*, and *kNN* benchmarks. C code was synthesised using Vivado HLS 2019.2 targeting a Xilinx Artix-7 FPGA. The objective function of the synthesis process considers the area to be as important as the delay because in real-world applications the energy used might be as important as the time that it takes to execute. The speedups varied all the way from 14× to 1392×. We evaluated the scalability of the backend in respect to the size of the input DFG and concluded that using the right configuration options, the execution time scales linearly with the size of the input DFG. The last section compared the synthesis results with those achieved by state-of-the-art approaches. Our framework proved to achieve outstanding results for the *SVM* and *dotprod* benchmarks and competitive results for the *kNN* benchmark. Being able to process the *kNN* benchmark is a significant step in order to be able to execute a whole new range of benchmarks. Further benchmark evaluations are needed to better understand the potential of the framework, but so far the results seem very promising.

# Chapter 6

# Conclusions

This chapter presents an overview of the approach, contributions provided, results achieved, and lastly, suggests some topics that should be handled by future work.

## 6.1 Concluding remarks

This dissertation proposes a new framework to generate efficient hardware implementations for a whole range of algorithms. This framework is two-stage, meaning that the first stage produces the input DFG, while the second one transforms this DFG into C code annotated with Vivado HLS directives. Although we have only used ANSI C benchmarks, the second stage is not strictly bound to any input language, as its input is a description of an execution trace, which can derive from any imperative high-level language (HLL). Moreover, as the transformations are independent of the target HLS tool, we can extend the framework to output code suitable for other HLS tools. Another advantage is that unlike some DSL approaches to the problem, this framework is not bond to any algorithm family, such as "machine learning" or "image processing". However, it performs best when there is high ILP to be exploited. The analysis of the complexity of the code generated shows that it is not trivial to manually replicate the transformations applied by the framework. The current work focuses on the extraction of operation-, data-, and task-level parallelism. However, task-level parallelism requires aggressive restructuring of the code to comply with the dataflow canonical guidelines. To exploit the use of the Vivado HLS *dataflow* directive, the framework extracts data-level parallelism by distributing data across multiple function calls, which are placed inside the dataflow region to be pipelined. Other contributions over the work proposed by Ferreira et al. [30] include more efficient pruning and leveling algorithms, support for simple control-flow and array accesses that depend on input data, and generation of code that leads to hardware with significant speedup improvements. Our experiments show results with speedups from 14× to 1392× when compared to the performance achieved by using the unmodified source code. When compared with state-of-the-art approaches, the framework proved to be competitive in all of the benchmarks. In conclusion, this framework fulfils its major objective, which is to enable developers to easily

generate efficient hardware implementations without the need for HLS expertise. However, there is still work to be done in order to increase the applicability of the approach.

## 6.2 Future work

Our future plans include the research of an efficient algorithm to cluster sequential isomorphic subgraphs, which would allow us to fold repetitive patterns of sequential operations. This is an essential transformation to be able to handle benchmarks, such as the kNN, in which the *Prologue* or *Epilogue* subgraphs grow at least linearly with the size of the input DFG. When the output code grows large it overuses resources and becomes unsynthesisable by any HLS tool. A possible solution to detect sequential subgraphs is to rerun the *Find All Subgraphs* algorithm in the *Prologue* and *Epilogue* graphs, but this time we would use the sequential weight of each node, which does not take into account the node level. However, this strategy would only give us lists of similar subgraphs, without any guarantee about their connection. Nevertheless, as the *Find All Subgraphs* algorithm performs a Breadth-First search, it would guarantee that a list of subgraphs with the same weight is ordered by their position in the graph from the top to the bottom. Thus, it would be trivial to check the subgraphs connections and find the largest chain of similar patterns. As the number of similar patterns might grow very large, it would be wise to use an heuristic to filter the most relevant ones. We intend to further research possible solutions to address the scalability problem. The current input DFG represents a full-trace of execution, therefore it does not have a way to represent loop constructions. Unfortunately, this can lead to graphs with millions of nodes which are unfeasible to parse. Possible solutions include having parameterized DFGs or providing user configurations that allow automatic scaling of the iteration space. However, the new DFG would no longer represent a full execution trace, which might in turn, blurry optimizations that were obvious otherwise. Thus, a balance needs to be considered. Finally, we intend to extend our framework with state-of-the-art analytical estimation models to refine the selection and configuration of directives. This can be achieved by integrating into our approach the *COMBA* model proposed by Zhao et al. [109], which is capable of finding a near-optimal configuration of multiple directives within minutes.

# Appendix A

# Benchmarks

This appendix provides the original code of each of the benchmarks.

## A.1 SVM

```
1  const float sv_coef[N_sv];
2  const float sup_vectors[D_sv][N_sv];
3
4  void svm_predict(float test_vector[D_sv], int *y){
5    float diff;
6    float norma;
7    float sum = 0;
8    for(int i=0; i< N_sv;i++){
9      norma=0;
10     for(int j=0; j<D_sv;j++){
11       diff=test_vector[j] - sup_vectors[j][i];
12       diff=diff*diff;
13       norma=norma + diff;
14     }
15     sum = sum + (exp(-gamma*norma)*sv_coeff[i]);
16   }
17   sum= sum-b;
18   *y = sum < 0 ? -1 : 1;
19 }
```

Listing A.1: SVM benchmark original prediction code

## A.2 Dotprod

```
1  int DSP_dotprod_golden_c(const short x[N], const short y[N])
2  {
3    int sum = 0, i;
```

```
4   for (i = 0; i < N; i++)
5     sum += x[i] * y[i];
6
7   return sum;
8 }
```

Listing A.2: Dot product original kernel

## A.3 kNN

```
1  #include "kNN.h"
2
3  /*
4   Update the set of k best points so far.
5   Points represent instances in the kNN model.
6   Those instances are the ones used in a previous learning phase.
7  */
8  void updateBest(dtype distance, ctype classifID, dtype BestPointsDistances[
       K], ctype BestPointsClasses[K]) {
9   dtype max = 0;
10  int index =0;
11
12  //find the worst point in the BestPoints
13  for(int i=0; i<K; i++) {
14   dtype dbest = BestPointsDistances[i];
15   dtype max_tmp = max;
16   max = (dbest > max_tmp) ? dbest : max;
17   index = (dbest > max_tmp) ? i : index;
18  }
19  // if the point is better (shorter distance) than the worst one (longer
        distance) in the BestPoints
20  // update BestPoints substituting the wrost one
21
22  dtype dbest = BestPointsDistances[index];
23  ctype cbest = BestPointsClasses[index];
24
25  BestPointsDistances[index] = (distance < max) ? distance : dbest;
26  BestPointsClasses[index] = (distance < max) ? classifID : cbest;
27 }
28
29 /**
30  kNN function without classifying but returning the k nearest points
31  We use here a linear search.
32 */
33 ctype kNN(ftype xFeatures[NUM_FEATURES], ftype knownFeatures[
       NUM_KNOWN_POINTS][NUM_FEATURES], ctype knownClasses[NUM_KNOWN_POINTS]) {
34
```

```
35  dtype BestPointsDistances[K]; // array with the distances of the K nearest
        points to the point to classify
36  ctype BestPointsClasses[K]; // array with the classes of the K nearest
        points to the point to classify
37
38
39  // initialize the data structures (array) with the K best points
40  initializeBest(BestPointsClasses, BestPointsDistances);
41
42
43  // perform the Euclidean distance between the point to classify and each
        one in the model
44  // and update the k best points if needed
45  for(int i=0; i<NUM_KNOWN_POINTS; i++) {
46   dtype distance = (dtype) 0;
47
48   // perform Euclidean distance
49   for(int j=0; j<NUM_FEATURES; j++) {
50    distance += sqr((dtype) xFeatures[j]-(dtype) knownFeatures[i][j]);
51   }
52   distance = sqrt(distance);
53   //printf("distance %e\n", distance);
54
55   // maintains the k best points updated
56   updateBest(distance, knownClasses[i], BestPointsDistances,
         BestPointsClasses);
57  }
58
59   // classify the point based on the K nearest points
60   ctype classifyID = classify3NN(BestPointsClasses, BestPointsDistances);
61
62     return classifyID;
63 }
64
65 /**
66  Classify based on the K BestPoints returned by the kNN function
67  Specialized code when using K = 3
68 */
69 ctype classify3NN(ctype BestPointsClasses[K], dtype BestPointsDistances[K])
       {
70
71   ctype c1 = BestPointsClasses[0];
72   dtype d1 = BestPointsDistances[0];
73
74   ctype c2 = BestPointsClasses[1];
75   dtype d2 = BestPointsDistances[1];
76
77   ctype c3 = BestPointsClasses[2];
78   dtype d3 = BestPointsDistances[2];
```

```
79
80    ctype classID;
81    dtype mindist = d1;
82
83    classID = (mindist > d2) ? c2 : c1;
84    mindist = (mindist > d2) ? d2 : d1;
85
86    classID = (mindist > d3) ? c3 : classID;
87    mindist = (mindist > d3) ? d3 : mindist;
88
89    classID = (c2 == c3) ? c2 : classID;
90    classID = (c1 == c3) ? c1 : classID;
91    classID = (c1 == c2) ? c1 : classID;
92
93
94   return classID;
95 }
```

Listing A.3: kNN original benchmark

# Appendix B

# Framework output

This appendix provides the output of the framework for each benchmark. Some parts of the code are omitted with "..." to avoid having dozens of pages per benchmark.

## B.1 SVM

```
1  void parallel_0(float sup_vectors[18][80], float test_vector[18], float
       sv_coeff[80], int width, float temp_l111_2_i2_array[80])
2  {
3    // Step 2: Initialize local variables
4    float diff_w1;
5    float norma_w1;
6    float temp_l111_1_i1_w1;
7    ...
8    // Initialization done
9    // starting Loop
10   for (int i = 0; i < width; i = i + 1) {
11   #pragma HLS pipeline
12
13     diff_w16 = test_vector[9] - sup_vectors[9][i];
14     diff_w10 = test_vector[8] - sup_vectors[8][i];
15     diff_w18 = test_vector[17] - sup_vectors[17][i];
16     diff_w9 = test_vector[10] - sup_vectors[10][i];
17     diff_w4 = test_vector[2] - sup_vectors[2][i];
18     diff_w11 = test_vector[5] - sup_vectors[5][i];
19     diff_w12 = test_vector[11] - sup_vectors[11][i];
20     diff_w14 = test_vector[1] - sup_vectors[1][i];
21     diff_w5 = test_vector[14] - sup_vectors[14][i];
22     diff_w17 = test_vector[15] - sup_vectors[15][i];
23     diff_w7 = test_vector[12] - sup_vectors[12][i];
24     diff_w13 = test_vector[13] - sup_vectors[13][i];
25     diff_w1 = test_vector[7] - sup_vectors[7][i];
26     diff_w8 = test_vector[0] - sup_vectors[0][i];
27     diff_w15 = test_vector[3] - sup_vectors[3][i];
```

```
28      diff_w3 = test_vector[6] - sup_vectors[6][i];
29      diff_w2 = test_vector[4] - sup_vectors[4][i];
30      diff_w6 = test_vector[16] - sup_vectors[16][i];
31
32      norma_w14 = 0 + square(diff_w8);
33      norma_w4 = norma_w14 + square(diff_w14);
34      norma_w15 = norma_w4 + square(diff_w4);
35      norma_w2 = norma_w15 + square(diff_w15);
36      norma_w11 = norma_w2 + square(diff_w2);
37      norma_w3 = norma_w11 + square(diff_w11);
38      norma_w1 = norma_w3 + square(diff_w3);
39      norma_w9 = norma_w1 + square(diff_w1);
40      norma_w16 = norma_w9 + square(diff_w10);
41      norma_w8 = norma_w16 + square(diff_w16);
42      norma_w12 = norma_w8 + square(diff_w9);
43      norma_w7 = norma_w12 + square(diff_w12);
44      norma_w13 = norma_w7 + square(diff_w7);
45      norma_w5 = norma_w13 + square(diff_w13);
46      norma_w17 = norma_w5 + square(diff_w5);
47      norma_w6 = norma_w17 + square(diff_w17);
48      norma_w18 = norma_w6 + square(diff_w6);
49      norma_w10 = norma_w18 + square(diff_w18);
50      temp_l111_1_i1_w1 = norma_w10 * (-gamma);
51      temp_l111_2_i2_array[i] = sv_coeff[i] * expf(temp_l111_1_i1_w1);
52    }
53  }
54
55  void epilogue(float temp_l111_2_i2_array_15[80], float
        temp_l111_2_i2_array_14[80], float temp_l111_2_i2_array_12[80], float
        temp_l111_2_i2_array_10[80], float temp_l111_2_i2_array_7[80], float
        temp_l111_2_i2_array_1[80], float temp_l111_2_i2_array_3[80], float
        temp_l111_2_i2_array_11[80], float temp_l111_2_i2_array_0[80], float
        temp_l111_2_i2_array_9[80], float temp_l111_2_i2_array_6[80], float
        temp_l111_2_i2_array_8[80], float temp_l111_2_i2_array_2[80], float
        temp_l111_2_i2_array_13[80], float temp_l111_2_i2_array_4[80], float
        temp_l111_2_i2_array_5[80], int *y)
56  {
57    // Initialize local variables
58    double operationOutput_w1;
59    float sum_w1;
60    float sum_w2;
61    float sum_w3;
62    ...
63    // Initialization done
64    sum_w787 = temp_l111_2_i2_array_3[7] + temp_l111_2_i2_array_3[8];
65    sum_w672 = temp_l111_2_i2_array_8[29] + temp_l111_2_i2_array_8[30];
66    sum_w368 = temp_l111_2_i2_array_4[51] + temp_l111_2_i2_array_4[52];
67    ...
68    sum_w1191 = sum_w5 + sum_w6;
```

```
69    sum_w1189 = sum_w295 + sum_w296;
70    sum_w437 = sum_w692 + sum_w693;
71    ...
72    sum_w669 = sum_w1159 - b;
73    operationOutput_w1 = sum_w669 < 0;
74
75    *y = operationOutput_w1 ? -1 : 1;
76  }
77
78  void svmResult(int *y)
79  {
80    // Step 2: Initialize local variables
81    float test_vector[18];
82    float sup_vectors_0[18][80];
83    ...
84    float sup_vectors_15[18][80];
85    float sv_coeff_0[80];
86    ...
87    float sv_coeff_15[80];
88    float temp_l111_2_i2_array_0[80];
89    ...
90    float temp_l111_2_i2_array_15[80];
91    // Initialization done
92    #pragma HLS dataflow
93
94    parallel_0(sup_vectors_0, test_vector, sv_coeff_0, 80,
          temp_l111_2_i2_array_0);
95    parallel_0(sup_vectors_1, test_vector, sv_coeff_1, 80,
          temp_l111_2_i2_array_1);
96    parallel_0(sup_vectors_2, test_vector, sv_coeff_2, 80,
          temp_l111_2_i2_array_2);
97    parallel_0(sup_vectors_3, test_vector, sv_coeff_3, 80,
          temp_l111_2_i2_array_3);
98    parallel_0(sup_vectors_4, test_vector, sv_coeff_4, 80,
          temp_l111_2_i2_array_4);
99    parallel_0(sup_vectors_5, test_vector, sv_coeff_5, 80,
          temp_l111_2_i2_array_5);
100   parallel_0(sup_vectors_6, test_vector, sv_coeff_6, 80,
          temp_l111_2_i2_array_6);
101   parallel_0(sup_vectors_7, test_vector, sv_coeff_7, 80,
          temp_l111_2_i2_array_7);
102   parallel_0(sup_vectors_8, test_vector, sv_coeff_8, 80,
          temp_l111_2_i2_array_8);
103   parallel_0(sup_vectors_9, test_vector, sv_coeff_9, 80,
          temp_l111_2_i2_array_9);
104   parallel_0(sup_vectors_10, test_vector, sv_coeff_10, 80,
          temp_l111_2_i2_array_10);
105   parallel_0(sup_vectors_11, test_vector, sv_coeff_11, 80,
          temp_l111_2_i2_array_11);
```

```
106   parallel_0(sup_vectors_12, test_vector, sv_coeff_12, 80,
          temp_l111_2_i2_array_12);
107   parallel_0(sup_vectors_13, test_vector, sv_coeff_13, 80,
          temp_l111_2_i2_array_13);
108   parallel_0(sup_vectors_14, test_vector, sv_coeff_14, 80,
          temp_l111_2_i2_array_14);
109   parallel_0(sup_vectors_15, test_vector, sv_coeff_15, 74,
          temp_l111_2_i2_array_15);
110
111   epilogue(temp_l111_2_i2_array_15, temp_l111_2_i2_array_14,
112        temp_l111_2_i2_array_12, temp_l111_2_i2_array_10,
               temp_l111_2_i2_array_7,
113        temp_l111_2_i2_array_1, temp_l111_2_i2_array_3,
               temp_l111_2_i2_array_11,
114        temp_l111_2_i2_array_0, temp_l111_2_i2_array_9,
               temp_l111_2_i2_array_6,
115        temp_l111_2_i2_array_8, temp_l111_2_i2_array_2,
               temp_l111_2_i2_array_13,
116        temp_l111_2_i2_array_4, temp_l111_2_i2_array_5, y);
117 }
```

Listing B.1: Framework output for the SVM benchmark with 15 parallel calls.

## B.2 Dotprod

```
1
2  void parallel_0(short x[256], short y[256], int sum_array[16])
3  {
4    // Step 2: Initialize local variables
5    int sum_w1;
6    ...
7    int sum_w14;
8
9    int temp_l83_i14_w1;
10   ...
11   int temp_l83_i29_w1;
12   // Initialization done
13   // starting Loop
14   for (int i = 0; i < 16; i = i + 1) {
15   #pragma HLS pipeline
16
17     temp_l83_i14_w1 = x[(16) * i] * y[(16) * i];
18     temp_l83_i15_w1 = x[(16) * i + 1] * y[(16) * i + 1];
19     temp_l83_i16_w1 = x[(16) * i + 2] * y[(16) * i + 2];
20     temp_l83_i17_w1 = x[(16) * i + 3] * y[(16) * i + 3];
21     temp_l83_i18_w1 = x[(16) * i + 4] * y[(16) * i + 4];
22     temp_l83_i19_w1 = x[(16) * i + 5] * y[(16) * i + 5];
```

```
23      temp_l83_i20_w1 = x[(16) * i + 6] * y[(16) * i + 6];
24      temp_l83_i21_w1 = x[(16) * i + 7] * y[(16) * i + 7];
25      temp_l83_i22_w1 = x[(16) * i + 8] * y[(16) * i + 8];
26      temp_l83_i23_w1 = x[(16) * i + 9] * y[(16) * i + 9];
27      temp_l83_i24_w1 = x[(16) * i + 10] * y[(16) * i + 10];
28      temp_l83_i25_w1 = x[(16) * i + 11] * y[(16) * i + 11];
29      temp_l83_i26_w1 = x[(16) * i + 12] * y[(16) * i + 12];
30      temp_l83_i27_w1 = x[(16) * i + 13] * y[(16) * i + 13];
31      temp_l83_i28_w1 = x[(16) * i + 14] * y[(16) * i + 14];
32      temp_l83_i29_w1 = x[(16) * i + 15] * y[(16) * i + 15];
33
34      sum_w13 = temp_l83_i14_w1 + temp_l83_i15_w1;
35      sum_w14 = temp_l83_i16_w1 + temp_l83_i17_w1;
36      sum_w11 = temp_l83_i18_w1 + temp_l83_i19_w1;
37      sum_w12 = temp_l83_i20_w1 + temp_l83_i21_w1;
38      sum_w1 = temp_l83_i22_w1 + temp_l83_i23_w1;
39      sum_w2 = temp_l83_i24_w1 + temp_l83_i25_w1;
40      sum_w5 = temp_l83_i26_w1 + temp_l83_i27_w1;
41      sum_w6 = temp_l83_i28_w1 + temp_l83_i29_w1;
42
43      sum_w3 = sum_w1 + sum_w2;
44      sum_w8 = sum_w11 + sum_w12;
45      sum_w7 = sum_w13 + sum_w14;
46      sum_w4 = sum_w5 + sum_w6;
47
48      sum_w10 = sum_w3 + sum_w4;
49      sum_w9 = sum_w7 + sum_w8;
50
51      sum_array[i] = sum_w9 + sum_w10;
52    }
53  }
54
55  void parallel_1(short x[192], short y[192], int sum_array[12])
56  {
57    // Step 2: Initialize local variables
58    int sum_w1;
59    ...
60    int sum_w14;
61
62    int temp_l83_i1806_w1;
63    ...
64    int temp_l83_i1821_w1;
65    // Initialization done
66    // starting Loop
67    for (int i = 0; i < 12; i = i + 1) {
68    #pragma HLS pipeline
69
70      temp_l83_i1806_w1 = x[(16) * i] * y[(16) * i];
71      temp_l83_i1807_w1 = x[(16) * i + 1] * y[(16) * i + 1];
```

```
72      temp_l83_i1808_w1 = x[(16) * i + 2] * y[(16) * i + 2];
73      temp_l83_i1809_w1 = x[(16) * i + 3] * y[(16) * i + 3];
74      temp_l83_i1810_w1 = x[(16) * i + 4] * y[(16) * i + 4];
75      temp_l83_i1811_w1 = x[(16) * i + 5] * y[(16) * i + 5];
76      temp_l83_i1812_w1 = x[(16) * i + 6] * y[(16) * i + 6];
77      temp_l83_i1813_w1 = x[(16) * i + 7] * y[(16) * i + 7];
78      temp_l83_i1814_w1 = x[(16) * i + 8] * y[(16) * i + 8];
79      temp_l83_i1815_w1 = x[(16) * i + 9] * y[(16) * i + 9];
80      temp_l83_i1816_w1 = x[(16) * i + 10] * y[(16) * i + 10];
81      temp_l83_i1817_w1 = x[(16) * i + 11] * y[(16) * i + 11];
82      temp_l83_i1818_w1 = x[(16) * i + 12] * y[(16) * i + 12];
83      temp_l83_i1819_w1 = x[(16) * i + 13] * y[(16) * i + 13];
84      temp_l83_i1820_w1 = x[(16) * i + 14] * y[(16) * i + 14];
85      temp_l83_i1821_w1 = x[(16) * i + 15] * y[(16) * i + 15];
86
87      sum_w11 = temp_l83_i1806_w1 + temp_l83_i1807_w1;
88      sum_w12 = temp_l83_i1808_w1 + temp_l83_i1809_w1;
89      sum_w13 = temp_l83_i1810_w1 + temp_l83_i1811_w1;
90      sum_w14 = temp_l83_i1812_w1 + temp_l83_i1813_w1;
91      sum_w7 = temp_l83_i1814_w1 + temp_l83_i1815_w1;
92      sum_w8 = temp_l83_i1816_w1 + temp_l83_i1817_w1;
93      sum_w3 = temp_l83_i1818_w1 + temp_l83_i1819_w1;
94      sum_w4 = temp_l83_i1820_w1 + temp_l83_i1821_w1;
95
96      sum_w9 = sum_w11 + sum_w12;
97      sum_w10 = sum_w13 + sum_w14;
98      sum_w6 = sum_w3 + sum_w4;
99      sum_w5 = sum_w7 + sum_w8;
100
101     sum_w2 = sum_w5 + sum_w6;
102     sum_w1 = sum_w9 + sum_w10;
103
104     sum_array[i] = sum_w1 + sum_w2;
105   }
106 }
107
108 void epilogue(int sum_array_3[16], int sum_array_2[16], short y_9[3], int
        sum_array_5[16], short x_9[3], int sum_array_4[16], int sum_array_7[12],
         int sum_array_6[16], short x_0[13], short y_0[13], int sum_array_1[16],
         int sum_array_0[16], int *out)
109 {
110   // Step 2: Initialize local variables
111   int sum_w1;
112   ...
113   int sum_w139;
114
115   int temp_l83_i1_w1;
116   ...
117   int temp_l83_i2000_w1;
```

```
118
119   // Initialization done
120   sum_w137 = sum_array_0[0] + sum_array_0[1];
121   ...
122   temp_l83_i1_w1 = x_0[0] * y_0[0];
123   ...
124   temp_l83_i2000_w1 = x_9[2] * y_9[2];
125
126   sum_w123 = sum_w100 + sum_w101;
127   ...
128   sum_w79 = sum_w104 + sum_w105;
129
130   *out = sum_w79 + temp_l83_i2000_w1;
131 }
132
133 void dotprod_parallel8(short x_0[13], short x_1[256], short x_2[256], short
        x_3[256], short x_4[256], short x_5[256], short x_6[256], short x_7
       [256], short x_8[192], short x_9[3], short y_0[13], short y_1[256],
       short y_2[256], short y_3[256], short y_4[256], short y_5[256], short
       y_6[256], short y_7[256], short y_8[192], short y_9[3], int *out) {
134   // Step 2: Initialize local variables
135   int sum_array_0[16];
136   int sum_array_1[16];
137   int sum_array_2[16];
138   int sum_array_3[16];
139   int sum_array_4[16];
140   int sum_array_5[16];
141   int sum_array_6[16];
142   int sum_array_7[12];
143 #pragma HLS ARRAY_PARTITION variable = x_0 cyclic factor = 13 dim = 0
144 #pragma HLS ARRAY_PARTITION variable = x_1 cyclic factor = 16 dim = 0
145 #pragma HLS ARRAY_PARTITION variable = x_2 cyclic factor = 16 dim = 0
146 #pragma HLS ARRAY_PARTITION variable = x_3 cyclic factor = 16 dim = 0
147 #pragma HLS ARRAY_PARTITION variable = x_4 cyclic factor = 16 dim = 0
148 #pragma HLS ARRAY_PARTITION variable = x_5 cyclic factor = 16 dim = 0
149 #pragma HLS ARRAY_PARTITION variable = x_6 cyclic factor = 16 dim = 0
150 #pragma HLS ARRAY_PARTITION variable = x_7 cyclic factor = 16 dim = 0
151 #pragma HLS ARRAY_PARTITION variable = x_8 cyclic factor = 16 dim = 0
152 #pragma HLS ARRAY_PARTITION variable = x_9 cyclic factor = 3 dim = 0
153 #pragma HLS ARRAY_PARTITION variable = y_0 cyclic factor = 13 dim = 0
154 #pragma HLS ARRAY_PARTITION variable = y_1 cyclic factor = 16 dim = 0
155 #pragma HLS ARRAY_PARTITION variable = y_2 cyclic factor = 16 dim = 0
156 #pragma HLS ARRAY_PARTITION variable = y_3 cyclic factor = 16 dim = 0
157 #pragma HLS ARRAY_PARTITION variable = y_4 cyclic factor = 16 dim = 0
158 #pragma HLS ARRAY_PARTITION variable = y_5 cyclic factor = 16 dim = 0
159 #pragma HLS ARRAY_PARTITION variable = y_6 cyclic factor = 16 dim = 0
160 #pragma HLS ARRAY_PARTITION variable = y_7 cyclic factor = 16 dim = 0
161 #pragma HLS ARRAY_PARTITION variable = y_8 cyclic factor = 16 dim = 0
162 #pragma HLS ARRAY_PARTITION variable = y_9 cyclic factor = 3 dim = 0
```

```
#pragma HLS ARRAY_PARTITION variable = sum_array_0 cyclic factor = 2 dim = 0
#pragma HLS ARRAY_PARTITION variable = sum_array_1 cyclic factor = 2 dim = 0
#pragma HLS ARRAY_PARTITION variable = sum_array_2 cyclic factor = 2 dim = 0
#pragma HLS ARRAY_PARTITION variable = sum_array_3 cyclic factor = 2 dim = 0
#pragma HLS ARRAY_PARTITION variable = sum_array_4 cyclic factor = 2 dim = 0
#pragma HLS ARRAY_PARTITION variable = sum_array_5 cyclic factor = 2 dim = 0
#pragma HLS ARRAY_PARTITION variable = sum_array_6 cyclic factor = 2 dim = 0
#pragma HLS ARRAY_PARTITION variable = sum_array_7 cyclic factor = 2 dim = 0
// Initialization done
#pragma HLS dataflow

  parallel_0(x_1, y_1, sum_array_0);
  parallel_0(x_2, y_2, sum_array_1);
  parallel_0(x_3, y_3, sum_array_2);
  parallel_0(x_4, y_4, sum_array_3);
  parallel_0(x_5, y_5, sum_array_4);
  parallel_0(x_6, y_6, sum_array_5);
  parallel_0(x_7, y_7, sum_array_6);
  parallel_1(x_8, y_8, sum_array_7);

  epilogue(sum_array_3, sum_array_2, y_9, sum_array_5, x_9, sum_array_4,
      sum_array_7, sum_array_6, x_0, y_0, sum_array_1, sum_array_0, out);
}
```

Listing B.2: Framework output for the Dotprod benchmark with 8 parallel calls.

## B.3 kNN

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <float.h>

#define NUM_CLASSES 2
#define MAXDISTANCE DBL_MAX
#define sqr(x) ((x) * (x))

```

```
11 void parallel_0(double knownFeatures[4][32], double xFeatures[32], double
       distance_array[4])
12 {
13   // Step 2: Initialize local variables
14   float distance_w1;
15   ...
16   float distance_w32;
17   float temp_l77_i1_w1;
18   ...
19   float temp_l77_i32_w1;
20   // Initialization done
21   // starting Loop
22   for (int i = 0; i < 4; i = i + 1) {
23   #pragma HLS pipeline
24
25     temp_l77_i1_w1 = xFeatures[0] - knownFeatures[i][0];
26     temp_l77_i2_w1 = xFeatures[1] - knownFeatures[i][1];
27     temp_l77_i3_w1 = xFeatures[2] - knownFeatures[i][2];
28     temp_l77_i4_w1 = xFeatures[3] - knownFeatures[i][3];
29     temp_l77_i5_w1 = xFeatures[4] - knownFeatures[i][4];
30     temp_l77_i6_w1 = xFeatures[5] - knownFeatures[i][5];
31     temp_l77_i7_w1 = xFeatures[6] - knownFeatures[i][6];
32     temp_l77_i8_w1 = xFeatures[7] - knownFeatures[i][7];
33     temp_l77_i9_w1 = xFeatures[8] - knownFeatures[i][8];
34     temp_l77_i10_w1 = xFeatures[9] - knownFeatures[i][9];
35     temp_l77_i11_w1 = xFeatures[10] - knownFeatures[i][10];
36     temp_l77_i12_w1 = xFeatures[11] - knownFeatures[i][11];
37     temp_l77_i13_w1 = xFeatures[12] - knownFeatures[i][12];
38     temp_l77_i14_w1 = xFeatures[13] - knownFeatures[i][13];
39     temp_l77_i15_w1 = xFeatures[14] - knownFeatures[i][14];
40     temp_l77_i16_w1 = xFeatures[15] - knownFeatures[i][15];
41     temp_l77_i17_w1 = xFeatures[16] - knownFeatures[i][16];
42     temp_l77_i18_w1 = xFeatures[17] - knownFeatures[i][17];
43     temp_l77_i19_w1 = xFeatures[18] - knownFeatures[i][18];
44     temp_l77_i20_w1 = xFeatures[19] - knownFeatures[i][19];
45     temp_l77_i21_w1 = xFeatures[20] - knownFeatures[i][20];
46     temp_l77_i22_w1 = xFeatures[21] - knownFeatures[i][21];
47     temp_l77_i23_w1 = xFeatures[22] - knownFeatures[i][22];
48     temp_l77_i24_w1 = xFeatures[23] - knownFeatures[i][23];
49     temp_l77_i25_w1 = xFeatures[24] - knownFeatures[i][24];
50     temp_l77_i26_w1 = xFeatures[25] - knownFeatures[i][25];
51     temp_l77_i27_w1 = xFeatures[26] - knownFeatures[i][26];
52     temp_l77_i28_w1 = xFeatures[27] - knownFeatures[i][27];
53     temp_l77_i29_w1 = xFeatures[28] - knownFeatures[i][28];
54     temp_l77_i30_w1 = xFeatures[29] - knownFeatures[i][29];
55     temp_l77_i31_w1 = xFeatures[30] - knownFeatures[i][30];
56     temp_l77_i32_w1 = xFeatures[31] - knownFeatures[i][31];
57
58     distance_w27 = 0 + sqr(temp_l77_i1_w1);
```

```
59      distance_w4 = sqr(temp_l77_i28_w1) + sqr(temp_l77_i29_w1);
60      distance_w32 = sqr(temp_l77_i30_w1) + sqr(temp_l77_i31_w1);
61      distance_w24 = distance_w27 + sqr(temp_l77_i2_w1);
62      distance_w3 = distance_w4 + distance_w32;
63      distance_w23 = distance_w24 + sqr(temp_l77_i3_w1);
64      distance_w30 = distance_w23 + sqr(temp_l77_i4_w1);
65      distance_w22 = distance_w30 + sqr(temp_l77_i5_w1);
66      distance_w21 = distance_w22 + sqr(temp_l77_i6_w1);
67      distance_w19 = distance_w21 + sqr(temp_l77_i7_w1);
68      distance_w18 = distance_w19 + sqr(temp_l77_i8_w1);
69      distance_w16 = distance_w18 + sqr(temp_l77_i9_w1);
70      distance_w15 = distance_w16 + sqr(temp_l77_i10_w1);
71      distance_w29 = distance_w15 + sqr(temp_l77_i11_w1);
72      distance_w28 = distance_w29 + sqr(temp_l77_i12_w1);
73      distance_w14 = distance_w28 + sqr(temp_l77_i13_w1);
74      distance_w13 = distance_w14 + sqr(temp_l77_i14_w1);
75      distance_w12 = distance_w13 + sqr(temp_l77_i15_w1);
76      distance_w9 = distance_w12 + sqr(temp_l77_i16_w1);
77      distance_w8 = distance_w9 + sqr(temp_l77_i17_w1);
78      distance_w20 = distance_w8 + sqr(temp_l77_i18_w1);
79      distance_w7 = distance_w20 + sqr(temp_l77_i19_w1);
80      distance_w6 = distance_w7 + sqr(temp_l77_i20_w1);
81      distance_w17 = distance_w6 + sqr(temp_l77_i21_w1);
82      distance_w11 = distance_w17 + sqr(temp_l77_i22_w1);
83      distance_w10 = distance_w11 + sqr(temp_l77_i23_w1);
84      distance_w2 = distance_w10 + sqr(temp_l77_i24_w1);
85      distance_w1 = distance_w2 + sqr(temp_l77_i25_w1);
86      distance_w5 = distance_w1 + sqr(temp_l77_i26_w1);
87      distance_w31 = distance_w5 + sqr(temp_l77_i27_w1);
88      distance_w26 = distance_w31 + distance_w3;
89      distance_w25 = distance_w26 + sqr(temp_l77_i32_w1);
90      distance_array[i] = sqrtf(distance_w25);
91    }
92  }
93
94  void epilogue(char knownClasses[8], double distance_array_1[4], double
        distance_array_0[4], char *out)
95  {
96    // Step 2: Initialize local variables
97    char BestPointsClasses[3];
98    double BestPointsDistances[3];
99    char c1_w1;
100   char c2_w1;
101   char c3_w1;
102   char cbest_w1;
103   ...
104   char cbest_w8;
105
106   char classID_w1;
```

```
107   char classID_w2;
108   char classID_w3;
109   char classID_w4;
110
111   double d1_w1;
112   double d2_w1;
113   double d3_w1;
114
115   float dbest_w1;
116   ...
117   float dbest_w32;
118
119   int index_w1;
120   int index_w24;
121
122   double max_tmp_w1;
123   ...
124   double max_tmp_w24;
125
126   float max_w1;
127   ...
128   float max_w24;
129
130   double mindist_w1;
131   double mindist_w2;
132
133   double muxOutput_w1;
134   ...
135   double muxOutput_w16;
136
137   double operationOutput_w1;
138   ...
139   double operationOutput_w38;
140
141   // Initialization done
142   max_tmp_w15 = 0;
143   max_tmp_w8 = 0;
144   max_tmp_w13 = 0;
145   max_tmp_w6 = 0;
146   max_tmp_w24 = 0;
147   max_tmp_w19 = 0;
148   max_tmp_w1 = 0;
149   max_tmp_w4 = 0;
150   BestPointsDistances[2] = MAXDISTANCE;
151   BestPointsDistances[0] = MAXDISTANCE;
152   BestPointsDistances[1] = MAXDISTANCE;
153   BestPointsClasses[0] = NUM_CLASSES;
154   BestPointsClasses[1] = NUM_CLASSES;
155   BestPointsClasses[2] = NUM_CLASSES;
```

```
156
157   dbest_w18 = BestPointsDistances[0];
158   dbest_w32 = BestPointsDistances[1];
159   dbest_w23 = BestPointsDistances[2];
160
161   operationOutput_w21 = dbest_w18 > max_tmp_w4;
162
163   index_w7 = operationOutput_w21 ? 0 : 0;
164   max_w24 = operationOutput_w21 ? dbest_w18 : 0;
165
166   max_tmp_w22 = max_w24;
167
168   ...
169
170   muxOutput_w3 = operationOutput_w2 ? knownClasses[0] : cbest_w3;
171   muxOutput_w1 = operationOutput_w2 ? distance_array_0[0] : dbest_w2;
172
173   BestPointsClasses[index_w4] = muxOutput_w3;
174   BestPointsDistances[index_w4] = muxOutput_w1;
175
176   dbest_w9 = BestPointsDistances[0];
177   dbest_w5 = BestPointsDistances[1];
178   dbest_w27 = BestPointsDistances[2];
179
180   ...
181
182   c1_w1 = BestPointsClasses[0];
183   c2_w1 = BestPointsClasses[1];
184   c3_w1 = BestPointsClasses[2];
185   d1_w1 = BestPointsDistances[0];
186   d2_w1 = BestPointsDistances[1];
187   d3_w1 = BestPointsDistances[2];
188
189   mindist_w1 = d1_w1;
190   operationOutput_w6 = c1_w1 == c3_w1;
191   operationOutput_w8 = c1_w1 == c2_w1;
192   operationOutput_w14 = c2_w1 == c3_w1;
193
194   operationOutput_w5 = mindist_w1 > d2_w1;
195   operationOutput_w32 = mindist_w1 > d2_w1;
196
197   classID_w1 = operationOutput_w5 ? c2_w1 : c1_w1;
198   mindist_w2 = operationOutput_w32 ? d2_w1 : d1_w1;
199
200   operationOutput_w27 = mindist_w2 > d3_w1;
201
202   classID_w3 = operationOutput_w27 ? c3_w1 : classID_w1;
203
204   classID_w2 = operationOutput_w14 ? c2_w1 : classID_w3;
```

```
205
206   classID_w4 = operationOutput_w6 ? c1_w1 : classID_w2;
207
208   *out = operationOutput_w8 ? c1_w1 : classID_w4;
209 }
210
211 void kNN_8p32f_2parallel_saveEnergy(double xFeatures[32], char knownClasses
        [8], double knownFeatures_0[4][32], double knownFeatures_1[4][32], char
        *out)
212 {
213   // Step 2: Initialize local variables
214   double distance_array_0[4];
215   double distance_array_1[4];
216   #pragma HLS ARRAY_PARTITION variable = xFeatures cyclic factor = 32 dim =
          1
217   #pragma HLS ARRAY_PARTITION variable = knownFeatures_0 cyclic factor = 3
        dim = 2
218   #pragma HLS ARRAY_PARTITION variable = knownFeatures_1 cyclic factor = 3
        dim = 2
219   // Initialization done
220   #pragma HLS dataflow
221
222   parallel_0(knownFeatures_0, xFeatures, distance_array_0);
223   parallel_0(knownFeatures_1, xFeatures, distance_array_1);
224
225   epilogue(knownClasses, distance_array_1, distance_array_0, out);
226 }
```

Listing B.3: Framework output for the kNN benchmark (8 data points, 32 features) with 2 parallel calls.

```
1  #include <stdio.h>
2  #include <math.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <float.h>
6
7  #define NUM_CLASSES 2
8  #define MAXDISTANCE DBL_MAX
9  #define sqr(x) ((x) * (x))
10
11 void parallel_0(float knownFeatures[500][32], float xFeatures[32], float
       distance_array[500])
12 {
13   // Step 2: Initialize local variables
14   float distance_w1;
15   float distance_w10;
16   float distance_w11;
```

```
17    float distance_w12;
18    float distance_w13;
19    float distance_w14;
20    float distance_w15;
21    float distance_w16;
22    float distance_w17;
23    float distance_w18;
24    float distance_w19;
25    float distance_w2;
26    float distance_w20;
27    float distance_w21;
28    float distance_w22;
29    float distance_w23;
30    float distance_w24;
31    float distance_w25;
32    float distance_w26;
33    float distance_w27;
34    float distance_w28;
35    float distance_w29;
36    float distance_w3;
37    float distance_w30;
38    float distance_w31;
39    float distance_w4;
40    float distance_w5;
41    float distance_w6;
42    float distance_w7;
43    float distance_w8;
44    float distance_w9;
45    float temp_l77_i10_w1;
46    float temp_l77_i11_w1;
47    float temp_l77_i12_w1;
48    float temp_l77_i13_w1;
49    float temp_l77_i14_w1;
50    float temp_l77_i15_w1;
51    float temp_l77_i16_w1;
52    float temp_l77_i17_w1;
53    float temp_l77_i18_w1;
54    float temp_l77_i19_w1;
55    float temp_l77_i1_w1;
56    float temp_l77_i20_w1;
57    float temp_l77_i21_w1;
58    float temp_l77_i22_w1;
59    float temp_l77_i23_w1;
60    float temp_l77_i24_w1;
61    float temp_l77_i25_w1;
62    float temp_l77_i26_w1;
63    float temp_l77_i27_w1;
64    float temp_l77_i28_w1;
65    float temp_l77_i29_w1;
```

```
66    float temp_l77_i2_w1;
67    float temp_l77_i30_w1;
68    float temp_l77_i31_w1;
69    float temp_l77_i32_w1;
70    float temp_l77_i3_w1;
71    float temp_l77_i4_w1;
72    float temp_l77_i5_w1;
73    float temp_l77_i6_w1;
74    float temp_l77_i7_w1;
75    float temp_l77_i8_w1;
76    float temp_l77_i9_w1;
77    // Initialization done
78    // starting Loop
79    for (int i = 0; i < 500; i = i + 1)
80    {
81 #pragma HLS pipeline II = 3
82
83      temp_l77_i1_w1 = xFeatures[0] - knownFeatures[i][0];
84      temp_l77_i2_w1 = xFeatures[1] - knownFeatures[i][1];
85      temp_l77_i3_w1 = xFeatures[2] - knownFeatures[i][2];
86      temp_l77_i4_w1 = xFeatures[3] - knownFeatures[i][3];
87      temp_l77_i5_w1 = xFeatures[4] - knownFeatures[i][4];
88      temp_l77_i6_w1 = xFeatures[5] - knownFeatures[i][5];
89      temp_l77_i7_w1 = xFeatures[6] - knownFeatures[i][6];
90      temp_l77_i8_w1 = xFeatures[7] - knownFeatures[i][7];
91      temp_l77_i9_w1 = xFeatures[8] - knownFeatures[i][8];
92      temp_l77_i10_w1 = xFeatures[9] - knownFeatures[i][9];
93      temp_l77_i11_w1 = xFeatures[10] - knownFeatures[i][10];
94      temp_l77_i12_w1 = xFeatures[11] - knownFeatures[i][11];
95      temp_l77_i13_w1 = xFeatures[12] - knownFeatures[i][12];
96      temp_l77_i14_w1 = xFeatures[13] - knownFeatures[i][13];
97      temp_l77_i15_w1 = xFeatures[14] - knownFeatures[i][14];
98      temp_l77_i16_w1 = xFeatures[15] - knownFeatures[i][15];
99      temp_l77_i17_w1 = xFeatures[16] - knownFeatures[i][16];
100     temp_l77_i18_w1 = xFeatures[17] - knownFeatures[i][17];
101     temp_l77_i19_w1 = xFeatures[18] - knownFeatures[i][18];
102     temp_l77_i20_w1 = xFeatures[19] - knownFeatures[i][19];
103     temp_l77_i21_w1 = xFeatures[20] - knownFeatures[i][20];
104     temp_l77_i22_w1 = xFeatures[21] - knownFeatures[i][21];
105     temp_l77_i23_w1 = xFeatures[22] - knownFeatures[i][22];
106     temp_l77_i24_w1 = xFeatures[23] - knownFeatures[i][23];
107     temp_l77_i25_w1 = xFeatures[24] - knownFeatures[i][24];
108     temp_l77_i26_w1 = xFeatures[25] - knownFeatures[i][25];
109     temp_l77_i27_w1 = xFeatures[26] - knownFeatures[i][26];
110     temp_l77_i28_w1 = xFeatures[27] - knownFeatures[i][27];
111     temp_l77_i29_w1 = xFeatures[28] - knownFeatures[i][28];
112     temp_l77_i30_w1 = xFeatures[29] - knownFeatures[i][29];
113     temp_l77_i31_w1 = xFeatures[30] - knownFeatures[i][30];
114     temp_l77_i32_w1 = xFeatures[31] - knownFeatures[i][31];
```

```
115
116     distance_w26 = sqr(temp_l77_i6_w1) + sqr(temp_l77_i7_w1);
117     distance_w27 = sqr(temp_l77_i4_w1) + sqr(temp_l77_i5_w1);
118     distance_w28 = sqr(temp_l77_i2_w1) + sqr(temp_l77_i3_w1);
119     distance_w19 = sqr(temp_l77_i10_w1) + sqr(temp_l77_i11_w1);
120     distance_w4 = sqr(temp_l77_i28_w1) + sqr(temp_l77_i29_w1);
121     distance_w11 = sqr(temp_l77_i22_w1) + sqr(temp_l77_i23_w1);
122     distance_w14 = sqr(temp_l77_i12_w1) + sqr(temp_l77_i13_w1);
123     distance_w12 = sqr(temp_l77_i14_w1) + sqr(temp_l77_i15_w1);
124     distance_w10 = sqr(temp_l77_i16_w1) + sqr(temp_l77_i17_w1);
125     distance_w6 = sqr(temp_l77_i26_w1) + sqr(temp_l77_i27_w1);
126     distance_w30 = 0 + sqr(temp_l77_i1_w1);
127     distance_w21 = sqr(temp_l77_i20_w1) + sqr(temp_l77_i21_w1);
128     distance_w24 = sqr(temp_l77_i18_w1) + sqr(temp_l77_i19_w1);
129     distance_w20 = sqr(temp_l77_i8_w1) + sqr(temp_l77_i9_w1);
130     distance_w1 = sqr(temp_l77_i24_w1) + sqr(temp_l77_i25_w1);
131     distance_w3 = sqr(temp_l77_i30_w1) + sqr(temp_l77_i31_w1);
132
133     distance_w2 = distance_w4 + distance_w3;
134     distance_w5 = distance_w1 + distance_w6;
135     distance_w8 = distance_w21 + distance_w11;
136     distance_w13 = distance_w14 + distance_w12;
137     distance_w18 = distance_w20 + distance_w19;
138     distance_w9 = distance_w10 + distance_w24;
139     distance_w25 = distance_w27 + distance_w26;
140     distance_w29 = distance_w30 + distance_w28;
141
142     distance_w31 = distance_w5 + distance_w2;
143     distance_w7 = distance_w9 + distance_w8;
144     distance_w23 = distance_w18 + distance_w13;
145     distance_w22 = distance_w29 + distance_w25;
146
147     distance_w16 = distance_w22 + distance_w23;
148     distance_w17 = distance_w7 + distance_w31;
149
150     distance_w15 = distance_w16 + distance_w17;
151
152     distance_array[i] = distance_w15 + sqr(temp_l77_i32_w1);
153   }
154 }
155
156 void epilogue_0(char knownClasses[500], float distance_array[500], float
        BPD[3], char BPC[3])
157 {
158   // Step 2: Initialize local variables
159   char BestPointsClasses[3];
160   float BestPointsDistances[3];
161 #pragma HLS ARRAY_PARTITION variable = BestPointsClasses complete dim = 1
162 #pragma HLS ARRAY_PARTITION variable = BestPointsDistances complete dim = 1
```

```
163
164   BestPointsDistances[0] = MAXDISTANCE;
165   BestPointsDistances[2] = MAXDISTANCE;
166   BestPointsDistances[1] = MAXDISTANCE;
167   BestPointsClasses[1] = NUM_CLASSES;
168   BestPointsClasses[0] = NUM_CLASSES;
169   BestPointsClasses[2] = NUM_CLASSES;
170   float max;
171   int index;
172   float distance;
173   float dbest;
174   float max_tmp;
175   char cbest;
176   for (int pi0 = 0; pi0 < 500; pi0++)
177   {
178 #pragma HLS PIPELINE
179     max = 0;
180     index = 0;
181     distance = distance_array[pi0];
182
183     //find the worst point in the BestPoints
184     for (int i = 0; i < 3; i++)
185     {
186       dbest = BestPointsDistances[i];
187       max_tmp = max;
188       max = (dbest > max_tmp) ? dbest : max;
189       index = (dbest > max_tmp) ? i : index;
190     }
191     // if the point is better (shorter distance) than the worst one (longer
          distance) in the BestPoints
192     // update BestPoints substituting the wrost one
193
194     dbest = BestPointsDistances[index];
195     cbest = BestPointsClasses[index];
196
197     BestPointsDistances[index] = (distance < max) ? distance : dbest;
198     BestPointsClasses[index] = (distance < max) ? knownClasses[pi0] : cbest
          ;
199   }
200   BPD[0] = BestPointsDistances[0];
201   BPD[1] = BestPointsDistances[1];
202   BPD[2] = BestPointsDistances[2];
203   BPC[0] = BestPointsClasses[0];
204   BPC[1] = BestPointsClasses[1];
205   BPC[2] = BestPointsClasses[2];
206 }
207
208 void epilogue(float BPD0[3], char BPC0[3], float BPD1[3], char BPC1[3],
      char *out)
```

```
209 {
210   int bestDistances[6];
211   int bestClasses[6];
212 #pragma HLS ARRAY_PARTITION variable = bestDistances complete dim = 1
213 #pragma HLS ARRAY_PARTITION variable = bestClasses complete dim = 1
214   float d1 = MAXDISTANCE, d2 = MAXDISTANCE, d3 = MAXDISTANCE;
215   char c1, c2, c3;
216   for (int i = 0; i < 6; i++)
217   {
218     bestDistances[i] = BPD0[i];
219     bestDistances[i + 3] = BPD1[i];
220     bestClasses[i] = BPC0[i];
221     bestClasses[i + 3] = BPC1[i];
222   }
223
224   for (int j = 0; j < 6; j++)
225   {
226     if (bestDistances[j] < d1)
227     {
228       d3 = d2;
229       c3 = c2;
230       d2 = d1;
231       c2 = c1;
232       d1 = bestDistances[j];
233       c1 = bestClasses[j];
234     }
235     else if (bestDistances[j] < d2)
236     {
237       d3 = d2;
238       c3 = c2;
239       d2 = bestDistances[j];
240       c2 = bestClasses[j];
241     }
242     else if (bestDistances[j] < d3)
243     {
244       d3 = bestDistances[j];
245       c3 = bestClasses[j];
246     }
247   }
248
249   char classID = c1;
250   float mindist = d1;
251
252   classID = (c2 == c3) ? c2 : classID;
253
254   *out = classID;
255 }
256
257 void kNNFloatNoSqrt1000p32f_2parallel(
```

```
258     float xFeatures[32], char knownClasses0[500],
259     char knownClasses1[500], float knownFeatures_0[500][32],
260     float knownFeatures_1[500][32], char *out)
261 {
262   // Step 2: Initialize local variables
263   float distance_array_0[500];
264   float distance_array_1[500];
265   float BPD0[3];
266   float BPD1[3];
267   char BPC0[3];
268   char BPC1[3];
269 #pragma HLS ARRAY_PARTITION variable = BPD0 complete dim = 1
270 #pragma HLS ARRAY_PARTITION variable = BPD1 complete dim = 1
271 #pragma HLS ARRAY_PARTITION variable = BPC0 complete dim = 1
272 #pragma HLS ARRAY_PARTITION variable = BPC1 complete dim = 1
273 #pragma HLS ARRAY_PARTITION variable = xFeatures cyclic factor = 32 dim = 1
274 #pragma HLS ARRAY_PARTITION variable = knownFeatures_0 cyclic factor = 16
        dim = 2
275 #pragma HLS ARRAY_PARTITION variable = knownFeatures_1 cyclic factor = 16
        dim = 2
276 // Initialization done
277 #pragma HLS dataflow
278
279   parallel_0(knownFeatures_0, xFeatures, distance_array_0);
280   parallel_0(knownFeatures_1, xFeatures, distance_array_1);
281
282   epilogue_0(knownClasses0, distance_array_0, BPD0, BPC0);
283   epilogue_0(knownClasses1, distance_array_1, BPD1, BPC1);
284   epilogue(BPD0, BPC0, BPD1, BPC1, out);
285 }
```

Listing B.4: kNN benchmark with a manually written epilogue (1000 data points, 32 features) with 2 parallel calls.

# Appendix C

# Configuration file

This chapter contains an example of the configuration file that was used in the kNN benchmark.

```
{
  "inputs": [
    "xFeatures[32]",
    "knownFeatures[8][32]",
    "knownClasses[8]"
  ],
  "input_types": [
    "double",
    "double",
    "char"
  ],
  "outputs": [
    "*out"
  ],
  "output_types": [
    "char"
  ],
  "fold": true,
  "parallelizeSums": false,
  "arithmetic": true,
  "pruneLocalArrays": false,
  "graph": "kNN8p32f.dot",
  "outputFile": "kNN_output",
  "saveEnergy": true,
  "parallelFunctions": 2,
  "maxNodesPerSubgraph": 1000,
  "subgraphRepeats": 0,
  "minFoldLevels": 30,
  "maxFoldLevels": 70,
  "varsToPartition": [
    {
      "var": "knownFeatures",
      "dim": 0
```

```
    }
  ],
  "includes": [
    "<stdio.h>",
    "<math.h>",
    "<stdlib.h>",
    "<string.h>",
    "<float.h>"
  ],
  "defines": [
    "NUM_CLASSES 2",
    "MAXDISTANCE DBL_MAX",
    "sqr(x) ((x)*(x))"
  ]
}
```

Listing C.1: Sample configuration file for the kNN benchmark.

# Appendix D

# User Configurations

This appendix describes the user configuration options that allow controlling of the backend execution.

Table D.1: Examples and description of each mandatory configuration option.

| Configuration Option | Value | Brief Description |
|---|---|---|
| "inputs" | [<br>"test_vector[18]",<br>"sup_vectors[18][1274]",<br>"sv_coeff[1274]"<br>] | Input arguments of the top-level function.<br>If an input is an array, then it has only read accesses. |
| "input_types" | ["float","float","float"] | C types of each input. |
| "outputs" | [ "*y" ] | Output arguments of the top-level function.<br>Outputs are pointers or arrays which have write operations. |
| "output_types" | [ "int" ] | C types of each output. |
| "graph" | "svm1274.dot" | Path to the file that contains the input DFG. |
| "outputFile" | "svm1274parallel1" | Name of the file to be output.<br>The .c extension will be added automatically.<br>This name will also be used as the name of the top-level function. |

Table D.2: Examples and description of each optional configuration.

| Configuration Option | Value | Default Value | Brief Description |
|---|---|---|---|
| "fold" | true | false | If true, the graph will be restructured to use the dataflow directive. The backend will search for the best subgraph that can be executed in parallel and wrap it into a function that folds all parallel subgraphs into a loop. |
| "parallelizeSums" | true | false | If true, the ParallelizeSums algorithm will execute after the Pruning algorithm. |
| "arithmetic" | true | true | If true, the ArithmeticOptimisations algorithm will execute before printing the restructured code. |
| "pruneLocalArrays" | true | true | If true, it will replace all positions of local arrays by new scalar variables. Don't use if there are positions which can only be calculated in run-time. |
| "saveEnergy" | false | false | If true, it may reduce the array partition factors to generate more area-efficient implementations. |
| "parallelFunctions" | 1 | 1 | The number of calls to the function that can be executed in parallel. |
| "maxNodesPerSubgraph" | 1000 | 1000 | Upper bound on the number of nodes that can exist in each parallel subgraph. |
| "subgraphRepeats" | 0 | 0 | If 0, then it has no effect.<br>Otherwise, it sets the number of times that the parallel subgraphs repeat themselfs. |
| "minFoldLevels" | 1 | 1 | Sets the minimum number of levels of each parallel subgraph. This option has high impact in execution time. |
| "maxFoldLevels" | 100 | 100 | Sets the maximum number of levels of each parallel subgraph. This option has high impact in execution time. |
| "varsToPartition" | [<br>{"var": "sup_vectors",<br>"dim: 1},<br>{"var": "sv_coeff",<br>"dim: 0} ] | [ ] | Sets the vars and respective dimensions that should be split to allow for data-level parallelism. |
| "includes" | [ "<math.h>"] | [ ] | List of includes to be written in the output file. |
| "defines" | [ ] | [ ] | List of defines to be written in the output file. |

# References


[1] Miron Abramovici and Daniel Saab. Satisfiability on Reconfigurable Hardware. In Wayne Luk, Peter Y. K. Cheung, and Manfred Glesner, editors, *Field-Programmable Logic and Applications*, pages 448–456, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[2] Shereen Afifi, Hamid GholamHosseini, and Roopak Sinha. FPGA Implementations of SVM Classifiers: A Review. *SN Computer Science*, 1:1–17, 2020. doi:10.1007/s42979-020-00128-9.

[3] Arvind. Bluespec: A Language for Hardware Design, Simulation, Synthesis and Verification Invited Talk. In *Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE '03, page 249, USA, 2003. IEEE Computer Society.

[4] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, UC Berkley, USA, 2006.

[5] Peter J Ashenden. *The VHDL Cookbook*. Department of Computer Science, University of Adelaide, 1990.

[6] N. Azizi, I. Kuon, A. Egier, A. Darabiha, and P. Chow. Reconfigurable Molecular Dynamics Simulator. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 197–206, April 2004. doi:10.1109/FCCM.2004.48.

[7] David F. Bacon, Rodric Rabbah, and Sunil Shukla. FPGA Programming for the Masses. *Commun. ACM*, 56(4):56–63, April 2013. doi:10.1145/2436256.2436271.

[8] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 193–205. IEEE Press, 2019.

[9] I. J. Bertolacci, M. M. Strout, S. Guzik, J. Riley, and C. Olschanowsky. Identifying and Scheduling Loop Chains Using Directives. In *2016 Third Workshop on Accelerator Programming Using Directives (WACCPD)*, pages 57–67, Nov 2016. doi:10.1109/WACCPD.2016.010.

[10] João Bispo and João M.P. Cardoso. Clava: C/C++ source-to-source compilation using LARA. *SoftwareX*, 12:100565, 2020. doi:doi.org/10.1016/j.softx.2020.100565.

[11] Mihai Budiu, Pedro V. Artigas, and Seth Copen Goldstein. Dataflow: A Complement to Superscalar. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 177–186, Austin, TX, Mar 2005. URL: www.cs.cmu.edu/~seth/papers/budiu-ispass05.pdf.

[12] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The Garp Architecture and C Compiler. *Computer*, 33(4):62–69, April 2000. doi:10.1109/2.839323.

[13] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, page 33–36, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1950413.1950423.

[14] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen Brown, and Jason Anderson. LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13, 09 2013. doi:10.1145/2514740.

[15] João M. P. Cardoso, Pedro C. Diniz, Zlatko Petrov, Koen Bertels, Michael Hübner, Hans van Someren, Fernando Gonçalves, José Gabriel F. de Coutinho, George A. Constantinides, Bryan Olivier, Wayne Luk, Juergen Becker, Georgi Kuzmanov, Florian Thoma, Lars Braun, Matthias Kühnle, Razvan Nane, Vlad Mihai Sima, Kamil Krátký, José Carlos Alves, and João Canas Ferreira. REFLECT: Rendering FPGAs to Multi-core Embedded Computing. In João M. P. Cardoso and Michael Hübner, editors, *Reconfigurable Computing*, pages 261–289. Springer New York, New York, NY, 2011. doi:10.1007/978-1-4614-0061-5_11.

[16] João M. P. Cardoso and Markus Weinhardt. *High-Level Synthesis*, pages 23–47. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-319-26408-0_2.

[17] William Carter. A User Programmable Reconfigurable Gate Array. In *Proc. Custom Integrated Circuits Conf., May 1986*, 1986.

[18] Y. H. Cho and W. H. Mangione-Smith. Deep Packet Filter with Dedicated Logic and Read Only Memories. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 125–134, April 2004. doi:10.1109/FCCM.2004.25.

[19] N. Chugh, V. Vasista, S. Purini, and U. Bondhugula. A DSL Compiler for Accelerating Image Processing Pipelines on FPGAs. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 327–338, Sep. 2016. doi:10.1145/2967938.2967969.

[20] T. Cover and P. Hart. Nearest Neighbor Pattern Classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[21] Zefu Dai and Jianwen Zhu. Saturating the Transceiver Bandwidth: Switch Fabric Design on FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, page 67–76, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2145694.2145706.

[22] Eddie C. Davis, Michelle Mills Strout, and Catherine Olschanowsky. Transforming Loop Chains via Macro Dataflow Graphs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 265–277, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3168832.

[23] A. DeHon. Unifying Mesh- and Tree-Based Programmable Interconnect. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(10):1051–1065, Oct 2004. doi:10.1109/TVLSI.2004.834237.

[24] E. Del Sozzo, R. Baghdadi, S. Amarasinghe, and M. D. Santambrogio. A Unified Backend for Targeting FPGAs from DSLs. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 1–8, July 2018. doi:10.1109/ASAP.2018.8445108.

[25] The DOT Language. URL: graphviz.gitlab.io/_pages/doc/info/lang.html [cited 2020-01-23].

[26] Saar Drimer, Tim Güneysu, and Christof Paar. DSPs, BRAMs, and a Pinch of Logic: Extended Recipes for AES on FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 3(1), January 2010. doi:10.1145/1661438.1661441.

[27] Esam El-Araby, Tarek El-Ghazawi, J. Le Moigne, and K. Gaj. Wavelet Spectral Dimension Reduction of Hyperspectral Imagery on a Reconfigurable Computer. In *Proceedings. 2004 IEEE International Conference on Field- Programmable Technology (IEEE Cat. No.04EX921)*, pages 399–402, Dec 2004. doi:10.1109/FPT.2004.1393309.

[28] Tom Feist. Vivado Design Suite. *White Paper*, 5:30, 2012.

[29] Daniel A. P. L. Fernandes and João M. P. Cardoso. Accelerating Human Activity Recognition Systems on FPGAs through a DSL approach. In *FSP Workshop 2019; Sixth International Workshop on FPGAs for Software Programmers*, pages 1–8. VDE, 2019.

[30] A. C. Ferreira. Restructuring Software Code for High-Level Synthesis using a Graph-based Approach Targeting FPGAs. Master's thesis, MIEEC program, Faculty of Engineering of the University of Porto, Portugal, July 2018.

[31] A. C. Ferreira and João M. P. Cardoso. Graph-Based Code Restructuring Targeting HLS for FPGAs. In Christian Hochberger, Brent Nelson, Andreas Koch, Roger Woods, and Pedro Diniz, editors, *Applied Reconfigurable Computing*, pages 230–244, Cham, 01 2019. Springer International Publishing. doi:10.1007/978-3-030-17227-5_17.

[32] A. C. Ferreira and João M. P. Cardoso. Unfolding and Folding: a New Approach for Code Restructuring targeting HLS for FPGAs. In *FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers*, pages 1–10, Aug 2018.

[33] Jörg Fickenscher, Frank Hannig, and Jürgen Teich. DSL-Based Acceleration of Automotive Environment Perception and Mapping Algorithms for Embedded CPUs, GPUs, and FPGAs. In Martin Schoeberl, Christian Hochberger, Sascha Uhrig, Jürgen Brehm, and Thilo Pionteck, editors, *Architecture of Computing Systems – ARCS 2019*, pages 71–86, Cham, 2019. Springer International Publishing.

[34] Peter Flake, Phil Moorby, Steve Golson, Arturo Salz, and Simon Davidmann. Verilog HDL and Its Ancestors and Descendants. *Proc. ACM Program. Lang.*, 4(HOPL), June 2020. doi:10.1145/3386337.

[35] T. W. Fry and S. A. Hauck. SPIHT Image Compression on FPGAs. *IEEE Transactions on Circuits and Systems for Video Technology*, 15(9):1138–1147, Sep. 2005. doi:10.1109/TCSVT.2005.852625.

[36] M. Genovese and E. Napoli. ASIC and FPGA Implementation of the Gaussian Mixture Model Algorithm for Real-Time Segmentation of High Definition Video. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(3):537–547, March 2014. doi:10.1109/TVLSI.2013.2249295.

[37] Paul Graham and Brent E. Nelson. A Hardware Genetic Algorithm for the Travelling Salesman Problem on SPLASH 2. In *Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications*, FPL '95, page 352–361, Berlin, Heidelberg, 1995. Springer-Verlag.

[38] D. J. Greaves. Kiwi Scientific Acceleration at Large: Incremental Compilation and Multi-FPGA HLS Demo. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–1, Sep. 2017. doi:10.23919/FPL.2017.8056830.

[39] Bastian Hagedorn, Sergei Gorlatch, and Michel Steuwer. An Extension of a Functional Intermediate Language for Parallelizing Stencil Computations and its Optimizing GPU Implementation using OpenCL. Master's thesis, University of Münster, 2016.

[40] T. T. Hieu, T. N. Thinh, T. H. Vu, and S. Tomiyama. Optimization of Regular Expression Processing Circuits for NIDS on FPGA. In *2011 Second International Conference on Networking and Computing*, pages 105–112, Nov 2011. doi:10.1109/ICNC.2011.23.

[41] B. Hill, J. Smith, G. Srinivasa, K. Sonmez, A. Sirasao, A. Gupta, and M. Mukherjee. Precision Medicine and FPGA Technology: Challenges and Opportunities. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 655–658, 2017.

[42] D. T. Hoang. Searching Genetic Databases on SPLASH 2. In *[1993] Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–191, April 1993. doi:10.1109/FPGA.1993.279464.

[43] IEEE. IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-1987*, pages 1–218, 1988.

[44] IEEE. IEEE Standard Hardware Description Language Based on the Verilog(R) Hardware Description Language. *IEEE Std 1364-1995*, pages 1–688, Oct 1996. doi:10.1109/IEEESTD.1996.81542.

[45] IEEE. IEEE Standard Verilog Hardware Description Language. *IEEE Std 1364-2001*, pages 1–792, 2001.

[46] IEEE. IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language. *IEEE Std 1800-2005*, pages 1–648, 2005.

[47] IEEE. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pages 1–590, 2006.

[48] IEEE. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315, Feb 2018. doi:10.1109/IEEESTD.2018.8299595.

[49] IEEE. IEEE Standard for VHDL Language Reference Manual. *IEEE Std 1076-2019*, pages 1–673, 2019.

[50] Texas Instruments. SPRC265 TMS320c6000 DSP Library (DSPLIB) | TI.com. URL: www.ti.com/tool/SPRC265 [cited 2020-01-11].

[51] Intel. Stratix® IV FPGA Overview - Intel® FPGAs. URL: www.intel.com/content/www/us/en/products/programmable/fpga/stratix-iv.html [cited 2020-01-23].

[52] Preston A. Jackson, Cy P. Chan, Jonathan E. Scalera, Charles M. Rader, and M. Michael Vai. A Systolic FFT Architecture for Real Time FPGA Systems. Technical report, Massachusetts Institute of Technology Lincoln Laboratory, 2005.

[53] Arpith Jacob, Joseph Lancaster, Jeremy Buhler, Brandon Harris, and Roger D. Chamberlain. Mercury BLASTP: Accelerating Protein Sequence Alignment. *ACM Trans. Reconfigurable Technol. Syst.*, 1(2), June 2008. doi:10.1145/1371579.1371581.

[54] Ju-wook Jang, Seonil Choi, and V. K. K. Prasanna. Area and Time Efficient Implementations of Matrix Multiplication on FPGAs. In *2002 IEEE International Conference on Field-Programmable Technology, 2002. (FPT). Proceedings.*, pages 93–100, Dec 2002. doi:10.1109/FPT.2002.1188669.

[55] F. L. Kastensmidt, L. Sterpone, L. Carro, and M. S. Reorda. On the Optimal Design of Triple Modular Redundancy Logic for SRAM-based FPGAs. In *Design, Automation and Test in Europe*, pages 1290–1295 Vol. 2, March 2005. doi:10.1109/DATE.2005.229.

[56] Dirk Koch and Jim Torresen. FPGASort: A High Performance Sorting Architecture Exploiting Run-Time Reconfiguration on FPGAs for Large Problem Sorting. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, page 45–54, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1950413.1950427.

[57] Christopher D. Krieger, Michelle Mills Strout, Catherine Olschanowsky, Andrew Stone, Stephen Guzik, Xinfeng Gao, Carlo Bertolli, Paul H.J. Kelly, Gihan Mudalige, Brian Van Straalen, and Sam Williams. Loop Chaining: A Programming Abstraction for Balancing Locality and Parallelism. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pages 375–384, May 2013. doi:10.1109/IPDPSW.2013.68.

[58] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, USA, Dec 2002. *See* `llvm.cs.uiuc.edu`.

[59] Corinna Lee. UTDSP Benchmark Suite. URL: www.eecg.toronto.edu/~corinna/ [cited 2020-01-11].

[60] E. Lemoine and D. Merceron. Run Time Reconfiguration of FPGA for Scanning Genomic Databases. In *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'95)*, pages 90–98, April 1995. doi:10.1109/FPGA.1995.477414.

[61] P. H. W. Leong, M. P. Leong, O. Y. H. Cheung, T. Tung, C. M. Kwok, M. Y. Wong, and K. H. Lee. Pilchard — A Reconfigurable Computing Platform with Memory Slot Interface. In *Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '01, page 170–179, USA, 2001. IEEE Computer Society.

[62] J. C. Lyke, C. G. Christodoulou, G. A. Vera, and A. H. Edwards. An Introduction to Reconfigurable Systems. *Proceedings of the IEEE*, 103(3):291–317, March 2015. doi:10.1109/JPROC.2015.2397832.

[63] I. Mavroidis, I. Papaefstathiou, and D. Pnevmatikatos. A Fast FPGA-Based 2-Opt Solver for Small-Scale Euclidean Traveling Salesman Problem. In *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, pages 13–22, April 2007. doi:10.1109/FCCM.2007.40.

[64] Rob McCready. Real-Time Face Detection on a Configurable Hardware System. In Reiner W. Hartenstein and Herbert Grünbacher, editors, *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*, pages 157–162, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[65] Máire McLoone and John V. McCanny. Single-Chip FPGA Implementation of the Advanced Encryption Standard Algorithm. In Gordon Brebner and Roger Woods, editors, *Field-Programmable Logic and Applications*, pages 152–161, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[66] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert. HIPAcc: A Domain-Specific Language and Compiler for Image Processing. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):210–224, Jan 2016. doi:10.1109/TPDS.2015.2394802.

[67] Ashish Mishra, Mohit Agarwal, Abhijit Rameshwar Asati, and Kota Solomon Raju. Using Graph Isomorphism for Mapping of Data Flow Applications on Reconfigurable Computing Systems. *Microprocessors and Microsystems*, 51:343 – 355, 2017. doi:doi.org/10.1016/j.micpro.2016.12.008.

[68] Andrew Moore. *FPGAs For Dummies®, 2nd Intel® Special Edition*. John Wiley & Sons, Inc., 111 River St. Hoboken, 2 edition, 2017.

[69] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. PolyMage: Automatic Optimization for Image Processing Pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '15*, pages 429–443, Istanbul, Turkey, 2015. ACM Press. doi:10.1145/2694344.2694364.

[70] S. Murugan M., P. Kumar B., C. M. Ananda, and G. Lakshminarayanan. Design Approach for FPGA based High Bandwidth Fibre Channel Analyser for Aerospace Application. In *2019 4th International Conference on Electrical, Electronics, Communication, Computer Technologies and Optimization Techniques (ICEECCOT)*, pages 223–227, 2019.

[71] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, Oct 2016. doi:10.1109/TCAD.2015.2513673.

[72] J. H. Oh, Y. Hyun Yoon, J. K. Kim, H. Bin Ihm, S. H. Jeon, T. Heon Kim, and S. E. Lee. An fpga-based electronic control unit for automotive systems. In *2019 IEEE International Conference on Consumer Electronics (ICCE)*, pages 1–2, 2019.

[73] C. Olschanowsky, M. M. Strout, S. Guzik, J. Loffeld, and J. Hittinger. A Study on Balancing Parallelism, Data Locality, and Recomputation in Existing PDE Solvers. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 793–804, Nov 2014. doi:10.1109/SC.2014.70.

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

[75] D. A. Patterson. RAMP: research accelerator for multiple processors - a community vision for a shared experimental parallel HW/SW platform. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 1–, March 2006. doi:10.1109/ISPASS.2006.1620784.

[76] Peixin Zhong, M. Martonosi, P. Ashar, and S. Malik. Accelerating Boolean Satisfiability with Configurable Hardware. In *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251)*, pages 186–195, April 1998. doi:10.1109/FPGA.1998.707896.

[77] Russell J. Petersen and Brad Hutchings. An Assessment of the Suitability of FPGA-Based Systems for Use in Digital Signal Processing. In *Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications*, FPL '95, page 293–302, Berlin, Heidelberg, 1995. Springer-Verlag.

[78] Carl Adam Petri and Wolfgang Reisig. Petri Net. *Scholarpedia*, 3(4):6477, 2008.

[79] Farzin Piltan, Omid Avatefipour, Samira Soltani, Omid Mahmoudi, Mahmoud Reza, Safaei Nasrabad, Mehdi Eram, Zahra Esmaeili, Sara Heidari, Kamran Heidari, and Mohammad Ebrahimi. Design FPGA-Based CL-Minimum Control Unit. *International Journal of Hybrid Information Technology*, 9:101–118, 01 2016. doi:10.14257/ijhit.2016.9.1.10.

[80] H. Quinn and P. Graham. Terrestrial-Based Radiation Upsets: a Cautionary Tale. In *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pages 193–202, April 2005. doi:10.1109/FCCM.2005.61.

[81] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 48 of *PLDI '13*, page 519–530, New York, NY, USA, June 2013. Association for Computing Machinery. doi:10.1145/2491956.2462176.

[82] K. Ravindran, N. Satish, Yujia Jin, and K. Keutzer. An FPGA-Based Soft Multiprocessor System for IPv4 Packet Forwarding. In *International Conference on Field Programmable Logic and Applications, 2005.*, pages 487–492, Aug 2005. doi:10.1109/FPL.2005.1515769.

[83] Tiago Lascasas dos Santos. Acceleration of Applications with FPGA-based Computing Machines: Code Restructuring. Master's thesis, MIEIC program, Faculty of Engineering of the University of Porto, Portugal, July 2020.

[84] M. Shand and J. Vuillemin. Fast Implementations of RSA Cryptography. In *Proceedings of IEEE 11th Symposium on Computer Arithmetic*, pages 252–259, June 1993. doi:10.1109/ARITH.1993.378085.

[85] J. Shen, Y. Qiao, Y. Huang, M. Wen, and C. Zhang. Towards a Multi-array Architecture for Accelerating Large-scale Matrix Multiplication on FPGAs. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, May 2018. doi:10.1109/ISCAS.2018.8351474.

[86] Michael John Sebastian Smith. *Application-Specific Integrated Circuits*. Addison-Wesley Professional, 1st edition, 2008.

[87] Robert Stewart, Kirsty Duncan, Greg Michaelson, Paulo Garcia, Deepayan Bhowmik, and Andrew Wallace. RIPL: A Parallel Image Processing Language for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems*, 11(1):7:1–7:24, March 2018. doi:10.1145/3180481.

[88] G. Stitt, B. Grattan, J. Villarreal, and F. Vahid. Using On-Chip Configurable Logic to Reduce Embedded System Software Energy. In *Proceedings. 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 143–151, April 2002. doi:10.1109/FPGA.2002.1106669.

[89] S. Summers, A. Rose, and Peter Sanders. Using MaxCompiler for the High Level Synthesis of Trigger Algorithms. *Journal of Instrumentation*, 12:C02015–C02015, 02 2017. doi:10.1088/1748-0221/12/02/C02015.

[90] M Sussmann and T Hill. Intel HLS Compiler: Fast Design, Coding, and Hardware. In *White paper*. Intel, 2017.

[91] Keisuke Takano, Tetsuya Oda, and Masaki Kohata. Design of a DSL for Converting Rust Programming Language into RTL. In Leonard Barolli, Yoshihiro Okada, and Flora Amato, editors, *Advances in Internet, Data and Web Technologies*, pages 342–350, Cham, 2020. Springer International Publishing.

[92] Abdelfatah Tamimi, Omaima Al-Allaf, and Mohammad Alia. Real-Time Group Face-Detection for an Intelligent Class-Attendance System. *International Journal of Information Technology and Computer Science*, 7:66–73, 05 2015. doi:10.5815/ijitcs.2015.06.09.

[93] R. Tessier, K. Pocek, and A. DeHon. Reconfigurable Computing Architectures. *Proceedings of the IEEE*, 103(3):332–354, March 2015. doi:10.1109/JPROC.2014.2386883.

[94] Stephen M Steve Trimberger. Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology. *IEEE Solid-State Circuits Magazine*, 10(2):16–29, 2018.

[95] K. H. Tsoi, K. H. Lee, and P. H. W. Leong. A Massively Parallel RC4 Key Search Engine. In *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '02, page 13, USA, 2002. IEEE Computer Society.

[96] Vasileios Tsoutsouras, Konstantina Koliogeorgi, Sotirios Xydis, and Dimitrios Soudris. An Exploration Framework for Efficient High-Level Synthesis of Support Vector Machines: Case Study on ECG Arrhythmia Detection for Xilinx Zynq SoC. *Journal of Signal Processing Systems*, 88(2):127–147, Aug 2017. doi:10.1007/s11265-017-1230-1.

[97] K. D. Underwood and K. S. Hemmert. Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 219–228, April 2004. doi:10.1109/FCCM.2004.21.

[98] J. Villasenor, C. Jones, and B. Schoner. Video Communications using Rapidly Reconfigurable Hardware. *IEEE Transactions on Circuits and Systems for Video Technology*, 5(6):565–567, Dec 1995. doi:10.1109/76.475899.

[99] J. Villasenor, B. Schoner, Kang-Ngee Chia, C. Zapata, Hea Kim, C. Jones, S. Lansing, and Bill mangione smith. Configurable Computing Solutions for Automatic Target Recognition. In *FPGAs for Custom Computing Machines*, pages 70 – 79, 05 1996. doi:10.1109/FPGA.1996.564749.

[100] Yutaka Watanabe, Jinpil Lee, Kentaro Sano, Taisuke Boku, and Mitsuhisa Sato. Design and Preliminary Evaluation of OpenACC Compiler for FPGA with OpenCL and Stream Processing DSL. In *HPCAsia2020: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Workshops*, pages 10–16, 01 2020. doi:10.1145/3373271.3373274.

[101] Sewook Wee, Jared Casper, Njuguna Njoroge, Yuriy Teslyar, Daxia Ge, Christos Kozyrakis, and Kunle Olukotun. A practical FPGA-based framework for novel CMP research. In *Proceedings of the ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*, pages 116–125, 01 2007. doi:10.1145/1216919.1216936.

[102] Tao Wu, ShuGuo Li, and LiTian Liu. Fast RSA Decryption through High-Radix Scalable Montgomery Modular Multipliers. *Science China Information Sciences*, 58(6):1–16, 2015. doi:10.1007/s11432-014-5215-4.

[103] Xilinx. Spartan-3 FPGA Family Data Sheet, 2013. URL: www.xilinx.com/support/documentation/data_sheets/ds099.pdf [cited 2020-09-05].

[104] Xilinx. 7 Series FPGAs Data Sheet: Overview, February 2018. URL: www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf [cited 2020-05-09].

[105] Xilinx. Vivado Design Suite User Guide High-Level Synthesis, UG902, April 2018. URL: www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug902-vivado-high-level-synthesis.pdf [cited 2020-16-9].

[106] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. CHStone: A Benchmark Program Suite for Practical C-based High-Level Synthesis. In *2008 IEEE International Symposium on Circuits and Systems*, pages 1192–1195, May 2008. doi:10.1109/ISCAS.2008.4541637.

[107] Ali Mustafa Zaidi and David Greaves. Value State Flow Graph: A Dataflow Compiler IR for Accelerating Control-Intensive Code in Spatial Hardware. *ACM Trans. Reconfigurable Technol. Syst.*, 9(2), December 2015. doi:10.1145/2807702.

[108] G. L. Zhang, P. H. W. Leong, C. H. Ho, K. H. Tsoi, C. C. C. Cheung, D. . Lee, R. C. C. Cheung, and W. Luk. Reconfigurable Acceleration for Monte Carlo based Financial Simulation. In *Proceedings. 2005 IEEE International Conference on Field-Programmable Technology, 2005.*, pages 215–222, Dec 2005. doi:10.1109/FPT.2005.1568549.

[109] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He. Performance Modeling and Directives Optimization for High-Level Synthesis on FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(7):1428–1441, July 2020. doi:10.1109/TCAD.2019.2912916.