FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# Acceleration of Applications with FPGA-based Computing Machines: New DSL

**Daniel Alexandre Pimenta Lopes Fernandes**

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: João Manuel Paiva Cardoso, PhD

July 25, 2019

# Acceleration of Applications with FPGA-based Computing Machines: New DSL

**Daniel Alexandre Pimenta Lopes Fernandes**

Mestrado Integrado em Engenharia Informática e Computação

July 25, 2019

# Abstract

Data analytics is often tackled by machine learning techniques. The algorithms involved frequently need to deal with large datasets, leading to long execution times. This kind of applications usually operates in domains where performance is critical. Developers have thus begun exploring hardware accelerators, such as graphics processing units (GPUs) and field-programmable gate arrays (FPGAs), to improve performance.

FPGAs provide a promising solution for hardware acceleration, configuration after manufacturing and reprogrammability. They are suitable targets for data analytics domains, providing higher performance and energy efficiency, which is becoming increasingly important in many systems nowadays. However, programming FPGAs is a complex task, usually accomplished using hardware description languages (HDLs), which require hardware design expertise. Several tools have been developed to raise the abstraction level of FPGA programming such as Xilinx's Vivado HLS. However, developers still need to be familiar with the tool they are using and the hardware they are targeting, in order to achieve efficient FPGA implementations. Therefore, if software developers are to adopt them, additional efforts must be made in standardizing the programming model, providing users with higher levels of abstraction.

One possible solution to this problem is the use of domain specific languages (DSLs), which have been proposed in many application domains. DSLs have also been developed to address the machine learning domain applied to Big Data. However, most of them do not target FPGAs. This dissertation proposes a new DSL for data analytics to target FPGA-based systems. The DSL compiler presented generates C code synthesizable by Vivado HLS for FPGA execution. The code is enhanced with optimization directives to take advantage of these devices. The DSL is evaluated using a human activity recognition (HAR) case study, taking into account performance, resource usage and productivity. The results show that data analytics applications can effectively be accelerated on FPGAs using a DSL approach. The main contributions of this work are towards the FPGA and data analytics communities (especially the ones using machine learning), as the developed DSL allows a new range of applications to be executed on FPGAs, thus achieving some of their benefits.

# Resumo

A análise de dados é muitas vezes abordada utilizando algoritmos de aprendizagem computacional. Estes algoritmos têm frequentemente de lidar com *datasets* muito grandes, o que leva a tempos de execução demorados. Este tipo de aplicações opera normalmente em dominios onde o desempenho é crítico. Os programadores começaram então a explorar aceleradores, como as *graphics processing units* (GPUs) e os *field-programmable gate arrays* (FPGAs), para melhorar o desempenho.

As FPGAs constituem uma solução promissora para a aceleração em *hardware*, configuração após fabrico e reprogramabilidade. Estes dispositivos são plataformas alvo adequadas para domínios de análise de dados, possibilitando níveis de desempenho e eficiência energética melhores, o que é cada vez mais um importante aspeto a considerar nos sistemas da atualidade. No entanto, programar FPGAs é uma tarefa complicada, normalmente concretizada utilizando linguagens de descrição de *hardware* (HDLs), que requerem conhecimentos de desenho de *hardware*. Diversas ferramentas foram desenvolvidas para melhorar o nivel de abstração da programação em FPGAs, como por exemplo o Vivado HLS da Xilinx. No entanto, os programadores ainda necessitam de estar familiarizados com a ferramenta e com a plataforma alvo, de modo a obterem implementações em FPGA eficientes. Desta forma, para estes dispositivos serem adotados por mais utilizadores, o modelo de programação terá de evoluir, a fim de aumentar o nivel de abstração.

Uma solução possível para este problema é o uso de linguagens de domínio específico (DSLs), que foram propostas em diversas áreas. Diversas DSLs foram também desenvolvidas para o domínio de aprendizagem computacional aplicada a *Big Data*. No entanto, a grande maioria não permite a execução em FPGAs. Esta dissertação propõe uma nova DSL para a área da análise de dados focando a execução em FPGAs. O compilador da DSL apresentado gera código C sintetizável pela ferramenta Vivado HLS. O código é enriquecido com diretivas de otimização, de modo a tirar partido da arquitetura destes dispositivos. A DSL é avaliada utilizando um caso de estudo de um sistema de reconhecimento de atividade humana (HAR), tendo em conta desempenho, utilização de recursos e produtividade. Os resultados demonstram que as aplicações de análise de dados podem ser efetivamente aceleradas em FPGAs utilizando uma DSL. As principais contribuições deste trabalho são direcionadas às areas das FPGAs e da análise de dados (especialmente aquelas que utilizam aprendizagem computacional), uma vez que a DSL desenvolvida permite que um novo leque de aplicações sejam executadas em FPGAs, tirando assim partido de algumas das suas vantagens.

# Acknowledgements

First and foremost, I would like to thank my supervisor João Manuel Paiva Cardoso for his support throughout this work. His knowledge in the field provided great contributions to the outcome of this dissertation.

I would also like to thank the folks over at the SPeCS lab, who have helped me throughout this dissertation, both clearing up any doubts I had and just generally giving me advice regarding the work.

Finally, I would like to thank my family for their support, not only throughout these few months, but also throughout my life in general. I am certain I wouldn't be where I am today without them.

Daniel Fernandes

*"It would appear that we have reached the limits of what it is possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound pretty silly in 5 years."*

John von Neumann

# Contents

# List of Figures

LIST OF FIGURES

# List of Tables

# Abbreviations

| | |
|---|---|
| ASIC | Application-Specific Integrated Circuit |
| ASSP | Application-Specific Standard Product |
| AST | Abstract Syntax Tree |
| BRAM | Block Random Access Memory |
| CPU | Central Processing Unit |
| DSL | Domain-Specific Language |
| DSP | Digital Signal Processor |
| FF | Flip-Flop |
| FPGA | Field-Programmable Gate Array |
| GPU | Graphics Processing Unit |
| HDL | Hardware Description Language |
| HLS | High-Level Synthesis |
| IDE | Integrated Development Environment |
| IP | Intellectual Property |
| IR | Intermediate Representation |
| LUT | Lookup Table |
| SoC | System on a Chip |

# Chapter 1

# Introduction

This chapter introduces the dissertation. The context of the work is given first, followed by the motivation and objectives of the dissertation. The last section provides a general outline of this dissertation with a brief description of each chapter and its contents.

## 1.1 Context

Ever since John Von Neumann wrote his famous first draft of what came to be known as the Von Neumann architecture [1, 2], computers have been, for the most part, developed following the principles therein. In modern times, however, some domains are moving computation towards hardware accelerators due to the constant need for better performance [3]. These devices attempt to employ a different paradigm of computation, favoring high degrees of parallelism. The most classic example is the computer graphics domain, which gave rise to the graphics processing units (GPUs) so familiar to everyone today. These devices, named after the graphics domain, were built to accelerate graphics processing. However, many more domains are embracing these architectures nowadays to improve performance [4]. Additionally, other architectures such as field-programmable gate arrays (FPGAs) are becoming more popular as well [5, 6, 7]. The parallel nature of these devices makes them appealing to several domains where performance is critical [8, 9]. Moreover, they also provide better energy efficiency than conventional central-processing units (CPUs) [10]. This is a characteristic that is becoming increasingly important in modern computing system, especially in embedded domains.

Data analytics [11] is the process of drawing conclusions based on thorough examination of data. The concept is often attached to machine learning, as the latter provides mechanisms for computers to analyze data, learn from it and thus be capable of making decisions based on the experience attained. Machine learning algorithms [12] must often deal with large amounts of data and the computations performed by them can become very complex. Problems arise when strict timing constraints must be met, especially in real-time environments. The volume of data and the inherent complexity of the algorithms significantly affect their performance. However, many algorithms display a heavy amount of parallelism. They are therefore appealing to accelerators, such as FPGAs, as these devices can take advantage of such parallelism to improve performance. Experiments have shown that targeting machine learning algorithms towards FPGAs can lead to critical performance gains [13].

## 1.2 Motivation

FPGAs provide a potential solution to the problems inherent to data analytics. The performance associated with the machine learning algorithms used in this domain can be significantly improved with the use of these devices. However, FPGAs suffer from one key issue: programmability. FPGAs are usually programmed using hardware description languages (HDLs) [14] such as VHDL or Verilog. These languages are hard to master for even expert software developers because they require hardware expertise. Software developers are used to programming in high level languages such as C or Java as these languages provide strong abstractions that allow them to efficiently develop programs without the need to know the underlying hardware being targeted.

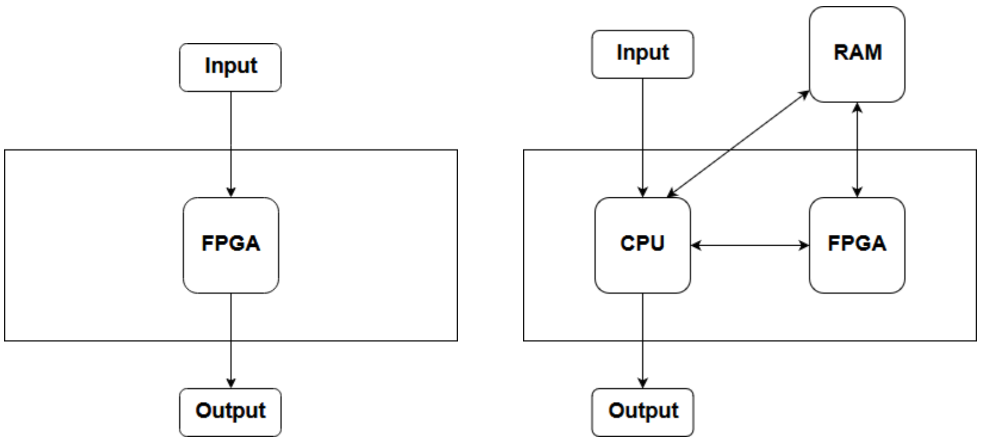
In an attempt to solve this problem, several parties have tried to improve the programmability of FPGAs, introducing high-level synthesis toolsets such as Xilinx's Vivado HLS [15] or Altera's OpenCL Compiler for FPGAs [16]. However, even these environments are difficult to use as they require developers to be familiar with the tool they are using and the hardware they are targeting. Multiple domain-specific languages (DSLs) have been proposed to target FPGAs (see, e.g., [17]). One of the most popular domains is image processing, as this is an area that can also benefit greatly from the use of accelerators. The restriction to a single domain allows DSL compilers to more efficiently produce code that can be targeted for FPGA execution, because the compiler can leverage its knowledge of the domain to perform domain-specific optimizations.

Multiple DSLs and frameworks exist in the machine learning domain applied to Big Data [18, 19]. However, most of these do not target FPGAs and the same holds for many other DSLs and frameworks in other domains. Programming FPGAs requires much more effort than programming CPUs or even GPUs. Ultimately, the FPGA programming model needs to improve, in order for these devices to be adopted by the community [20]. For now, however, DSLs remain a viable and elegant solution to specific problems.

## 1.3 Objectives

Targeting FPGAs can be done using two different approaches, illustrated in Figure 1.1. The first one uses the FPGA isolated. Inputs are sent in and out of the FPGA, so the entire program is hardware accelerated. The second approach involves a mixed CPU-FPGA solution. In this case, the CPU takes care of the main control flow of the program. The FPGA acts as a co-processor.

This dissertation proposes a new DSL for the data analytics domain targeting FPGA execution. The DSL provides machine learning oriented abstractions to allow users to naturally express their programs, hiding a number of implementation details. The main work includes the DSL compiler, responsible for generating optimized C code, synthesizable by the Vivado HLS toolset. This code



(a) Standalone FPGA scenario (b) Mixed CPU-FPGA scenario

Figure 1.1: Two different FPGA execution scenarios

can then be used to generate a bitstream for FPGA execution. Additionally, the DSL can also be used to program mixed CPU-FPGA systems, using Xilinx's SDSoC [21], as well as systems with a CPU only, using an ordinary C compiler. The DSL workflow is shown in Figure 1.2.

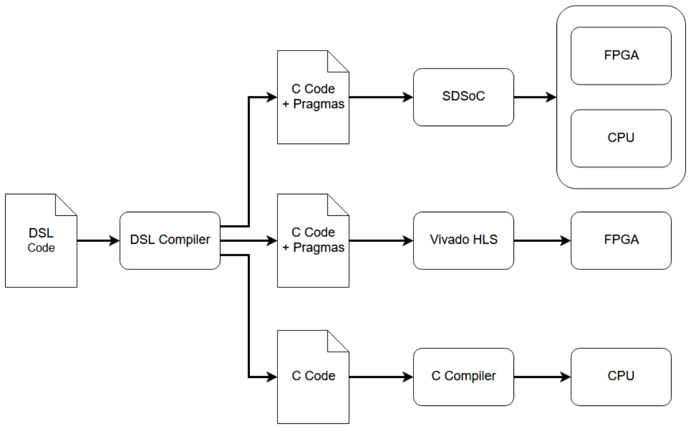


Figure 1.2: DSL workflow

The evaluation process of the developed DSL uses a human activity recognition (HAR) case study, taking into account both productivity and performance. For the performance evaluation, two scenarios are considered. The first one focuses on the FPGA code, in order to understand what sort of speedups can be obtained by targeting data analytics applications to a standalone FPGA. The second one considers a hybrid software/hardware solution, where the CPU manages the main control flow of the application, leaving the FPGA as a co-processor. The main goal of this work is to allow developers to create data analytics systems in a simplified manner and execute them on FPGAs, taking advantage of some of their benefits.

This dissertation's main contributions are targeted at the data analytics and FPGA communities. Data analytics users can benefit from a new DSL for the domain to target FPGA-based systems. Likewise, the introduction of a new DSL capable of targeting FPGAs improves the programmability of these devices, allowing a new range of applications to take advantage of them. As such, this dissertation aims to answer the following research questions:

A.1 *Can data analytics applications benefit from standalone FPGA execution using HLS?*

A.2 *Can data analytics applications benefit from mixed CPU-FPGA execution using HLS?*

B.1 *Can data analytics applications be targeted to FPGAs using a DSL approach?*

B.2 *Can data analytics applications improve performance using a new DSL for FPGAs?*

## 1.4 Outline

This dissertation is organized as follows. Chapter 1 introduces the dissertation, providing context and motivation for the work proposed. Chapter 2 provides background into field-programmable gate arrays and domain-specific languages. Chapter 3 reviews related work in domain-specific languages and frameworks, with an emphasis on machine learning and FPGAs. Chapter 4 dives into the DSL developed as part of the main work of this dissertation. Chapter 5 evaluates the DSL and Chapter 6 concludes this dissertation.

# Chapter 2

# Background

This chapter provides some background knowledge related to field-programmable gate arrays (FP-GAs) and domain-specific languages (DSLs).

## 2.1 Field-programmable Gate Arrays

Field-programmable gate arrays (FPGAs) are integrated circuits designed for configuration after manufacturing [10]. These circuits are composed as an array of logic blocks that can be configured by the user to perform combinational functions. The blocks can then be connected to each other with wires, thus allowing customers to create complex combinational circuits. FPGAs are usually programmed using hardware description languages (HDLs) such as Verilog or VHDL. Users can specify their circuits using these languages and use tools to synthesize bitstreams that can be fed to the FPGA fabric for reconfiguration.

FPGAs can provide hardware acceleration that increases performance over CPUs (see, e.g., [5, 13]). This happens because FPGAs emphasize parallelization and customization, allowing programs to be executed in a data-flow manner, as opposed to a control-oriented paradigm employed by traditional Von Neumann architectures [10].

### 2.1.1 Architecture

As mentioned earlier, FPGAs contain an array of logic blocks. The blocks are interconnected to each other with wires and input/output blocks allow data to be sent to and from the FPGA. The FPGA can therefore be seen as a grid of logic blocks, where each block can operate concurrently. This provides huge benefits in terms of performance, because work can be distributed across the FPGA fabric. Moreover, the fact that logic blocks can be connected to other blocks allows computations to be deeply pipelined [22]. An overview of an FPGA architecture is shown in Figure 2.1.

FPGAs employ a data-flow computation paradigm, because all logic blocks can operate at the same time. Therefore, despite having lower clock speeds than CPUs, these devices can achieve even better performance in certain situations. They are, however, always limited by Amdahl's

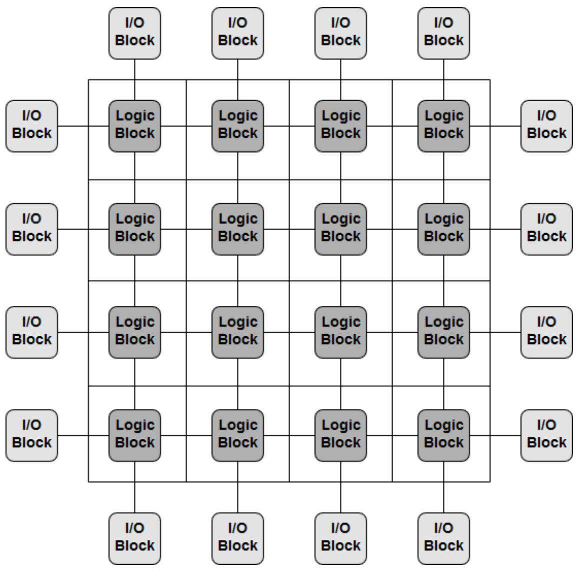


Figure 2.1: FPGA architecture

Law [23], as the amount of parallel work always needs to be substantial for significant performance gains to be achieved. If the problem at hand is highly sequential, a CPU will outperform an FPGA, because the FPGA will not be able to take advantage of its architecture. For this reason, FPGAs are commonly used as coprocessors, allowing highly parallel computations to be performed in the FPGA fabric, leaving any sequential work to a CPU. The FPGA therefore takes a role similar to that of a GPU, as the latter is also used as a coprocessor to perform parallel work.

Another key aspect that makes FPGAs so appealing is the mapping of instructions to hardware. As mentioned earlier, conventional processing units adopt the Von Neumann architecture principles, because they follow a control flow paradigm, using memory to store and fetch instructions for sequential execution. FPGAs, on the other hand, have no fetching of instructions because the instructions are mapped into the circuit itself. Figure 2.2 shows the basic idea. Each node in the computation graph can be a logic block (more than one logic block would be necessary for larger operations, of course). Due to the flexibility of the FPGA, each block runs concurrently. Note that if a certain operation is being performed multiple times, this type of architecture can pipeline the execution, improving performance. In the case of Figure 2.2, all the nodes in the graph can execute at the same time as long as the FPGA keeps receiving input values. For example, the last node in the graph can be computing the multiplication of the values it receives in step (clock cycle) $i$ while the nodes in the previous layer are executing the subtraction and multiplication for the values they receive in step $i + 1$. The last node would thus only need to wait 2 steps before receiving the first input values. After that, it would execute indefinitely.

This lack of instruction fetching and decoding allows FPGAs to achieve even better performance, because the circuit is tailored to a specific task. Of course, if a new algorithm is supposed to be executed on the FPGA, the fabric will need to be reconfigured. However, this is not a problem, because FPGAs were built to deal with specific tasks. If a system needs flexibility, a CPU is

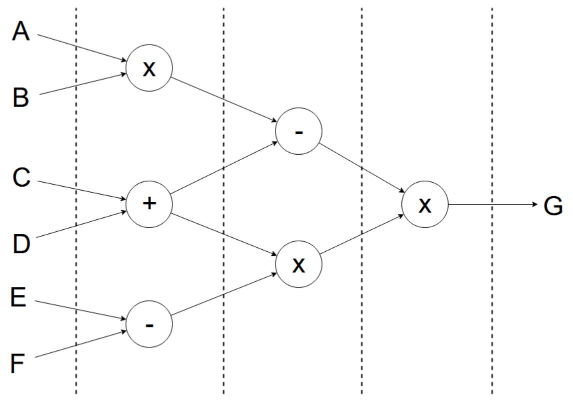


Figure 2.2: Computation graph of $G = ((A \times B) - (C + D)) \times ((C + D) \times (E - F))$

used instead. Some FPGA vendors provide dynamic reconfiguration features, but these add extra overhead and drive the FPGA into general purpose territory.

Each logic block usually contains a set of lookup tables (LUTs), flip-flops (FFs), digital signal processors (DSPs) and multiplexers. Lookup tables allow the user to store the functions desired while the flip-flops provide a mechanism for data to be persisted. The lookup tables can be configured with any logic function if the number of inputs in the LUT is enough. More complex functions might require the use of multiple LUTs or multiple logic blocks entirely. Current FPGAs also include on-chip memories distributed along the fabric [10].

### 2.1.2 History

FPGAs were developed in the 1980s to rival application-specific integrated circuits (ASICs). ASICs, unlike FPGAs, cannot be reconfigured. FPGAs presented an elegant solution over ASICs due to their reconfigurability. Although their performance was not quite as good as ASICs', their reconfigurability made them appealing to a wide range of markets. This rivalry still exists today, but both FPGAs and ASICs have evolved a lot since then [24]. Modern FPGAs have evolved into complex systems on a chip (SoCs) with several intellectual property (IP) cores attached to them [10]. IP cores can range from memory controllers to complete microprocessors. Nowadays, in fact, the reconfigurable part of the FPGA is usually less than half the entire fabric area. However, it is still the reconfigurable section that makes these devices so appealing. Application-specific standard products (ASSPs) have since showed up as well. These are like ASICs in that they provide specific features, but they tailor each solution to a wider audience.

### 2.1.3 Programming FPGAs

FPGAs are usually programmed using hardware description languages (HDLs). However, because HDLs are difficult to master for even experienced software developers, researchers and FPGA vendors started providing ways of programming these devices in more familiar languages such as

C. This process became known as high level synthesis (HLS) [25]. This section dives into both HDLs and HLS.

#### 2.1.3.1 Hardware Description Languages

Hardware description languages are programming languages that describe hardware. Despite having syntax very close to that of traditional general purpose programming languages, they differ in several other aspects. Programs written using HDLs are essentially logic circuits, using logic gates to express them. A key aspect that drifts HDLs apart from traditional programming languages is the notion of time. The most most popular HDLs available are Verilog and VHDL [14], although several others exist.

#### 2.1.3.2 High Level Synthesis

While hardware description languages might be very convenient to map to hardware, they are difficult to program in. Most software developers are not used to programming at such a low level. Researchers and FPGA vendors thus started focusing on the concept of high level synthesis (HLS) [25]. High level synthesis allows software developers to program FPGAs using familiar high level languages such as C. The synthesis process usually generates HDL code, which can then be used by FPGA tools to generate the bitstreams to configure the FPGA.

High level synthesis differs from traditional compilation processes in several ways. Mapping a program to hardware is radically different than what compilers traditionally do. In essence, the most important steps of high level synthesis are schedulling, allocation and binding [26, 27]. These are defined as follows:

- **Schedulling** - Assignment of operations to control steps;
- **Allocation** - Selection of resources used to implement the intended operations;
- **Binding** - Mapping of the operations to the hardware resources chosen during allocation.

All operations in a control step execute in the same clock cycle. It is thus important to maximize the number of operations executed in a control step to minimize the number of clock cycles on execution.

In addition to the three main synthesis tasks, optimizations are also somewhat different when synthesizing a hardware implementation. Even though certain optimizations, such as software pipelining or code transformations are also used in traditional compilers, high level synthesis allows additional hardware specific optimizations to be performed [25]. One common example is the use of custom bit-widths for variables in code. Traditionally, variables stick to strict variable types that have specific sizes. However, such sizes are not always adequate. For example, developers commonly use a 4 byte integer type for a loop counter, even when the loop runs very few times. When mapping to hardware, this conceives additional overhead. Therefore, high level synthesis can adjust the variables' sizes to be more efficient.

Several environments provide high level synthesis for FPGAs [28]. These range from commercial tools to academic projects. Most of them focus on synthesizing high-level languages such as C, C++ or Java.

Vivado High-Level Synthesis [15] is a high-level synthesis environment developed by Xilinx. It allows programs written in C, C++ and SystemC to be synthesized for FPGA execution. Vivado HLS comes with a full IDE as well as command line tools. This environment includes several wizards and scripts to help developers create their projects as well as synthesize their solutions. Vivado HLS also contains advanced features such as a debugger and an analysis perspective that allows synthesis results to be thoroughly examined. Code within Vivado HLS can be simulated, validated and synthesized as well as exported as an IP core. Moreover, the tool comes with a vast documentation with several examples and tutorials on each of the features described.

Vivado HLS C or C++ code is enhanced with optimization directives in the form of pragmas in the code. Directives can also be written to a separate file to provide developers with the opportunity to create multiple solutions in a project, each with its own directives. There are several directives available in Vivado HLS. Some of the most common compiler optimizations, such as loop pipelining and loop unrolling have their own directives. However, a number of Vivado HLS directives provide memory optimizations that are very specific to FPGAs.

While Vivado HLS allows the generation of IP cores, sometimes users want to develop mixed CPU-FPGA applications, using the FPGA as co-processor. Xilinx provides SDSoC [21] for these use cases. SDSoC allows applications to be executed on a mixed system consisting of a CPU and an FPGA. The application control flow is handled by the CPU, leaving parallel work to the FPGA. Note that this requires data transfers between CPU and FPGA every time an accelerated function is called. These transfers can be detrimental to the overall performance of the application. Therefore, SDSoC provides additional optimization directives targeted at communication. These allow the user to specify certain properties about the data transfers. For example, if an array is accessed in a sequential order, the accelerated function can begin executing as soon as it obtains the first element of the array. This access pattern can be configured using SDSoC optimization directives.

MaxCompiler [29] is a compiler developed by Maxeler Technologies that allows the generation of hardware implementations for execution on FPGAs. Developing an application requires users to create kernels, a manager wrapping those kernels and a host application [30]. The kernels and the manager are written in MaxJ, a Java-based high-level language that adds operator overloading [31]. The kernels represent the computations to be accelerated in hardware. The manager wraps the kernels and configures their interfaces to the host application. The host application is usually written in C or C++ and it launches the kernels using interface functions generated by the compiler. These functions are generated based on the kernel and manager code.

## 2.2 Domain-specific languages

Domain-specific languages (DSLs) are languages focused on a given domain [32]. They differ from general purpose programming languages in that they cannot handle a wide range of problems. Instead, a DSL chooses to focus a given application domain, providing high level abstractions that are useful to developers targeting that domain. Therefore, programs written in a domain-specific language are usually very simple to develop and easy to read. The DSL attempts to hide anything that is irrelevant to the domain itself, allowing the user to focus on the domain only.

One of the key advantages of using a DSL over a general purpose language is the opportunity to obtain significant performance gains. Since DSLs are explicitly focusing on a particular domain, compilers can take advantage of such domain knowledge to generate more efficient target implementations. Compilers are used to applying several optimizations to code generated, even in general purpose languages. However, a DSL compiler can take certain domain-specific optimizations that cannot be taken in a general-purpose context. This allows DSLs to achieve better performance than general purpose languages.

In the context of FPGAs, DSLs are an elegant solution to the problems associated with programming these devices. As mentioned in Section 2.1.3, programming FPGAs is usually done using either hardware description languages (HDLs) or high-level synthesis (HLS) tools. HDLs are very difficult to use for even experienced software programmers. FPGA vendors developed HLS tools in an attempt to solve this problem. However, these tools are still rather difficult to use. Developers still require some knowledge regarding the hardware they are targeting. Moreover, some familiarity with the tool itself is also required, in order to achieve efficient implementations. DSLs provide an interesting solution to this problem and they have had extensive use in several application domains targeting FPGAs [17]. These languages raise the level of abstraction on FPGAs because they rely on the compiler to deal with the FPGA specific issues, allowing the user to focus solely on the domain itself.

## 2.3 Summary

This chapter provided an overview of the necessary background for this dissertation. The architecture, history and programmability of FPGAs was explained and the concept of DSLs was also touched upon, in order to understand their relevance, especially for the FPGA domain.

It is clear that a DSL is a valid approach to program FPGAs, especially for developers unfamiliar with these devices. The learning curve of both HDLs and HLS environments drives most software developers away from FPGAs, but DSLs can certainly be useful in fighting this issue, allowing these devices to be used by more people. The adoption of FPGAs can benefit several domains, as more applications can leverage the parallel nature of these devices to improve performance. This is especially relevant for real time systems where strict timing constraints must be met.

# Chapter 3

# Related Work

This chapter overviews related work in domain-specific languages (DSLs), focusing on accelerators (GPUs and FPGAs) and machine learning. Furthermore, some machine learning frameworks are also explored as these can provide interesting domain-specific abstractions as well.

## 3.1 DSLs targeting accelerators

This section details some of the most relevant work done in DSLs targeting accelerators. Most of these DSLs target GPUs, but some approaches for FPGA execution are also described.

### 3.1.1 HIPA$^{CC}$

HIPA$^{CC}$ [33] is a framework that allows users to develop image processing algorithms, generating code targeting embedded GPUs. HIPA$^{CC}$ uses a C++-based embedded domain-specific language to specify image processing pipelines with useful abstractions. The HIPA$^{CC}$ compiler then generates the necessary code for different target languages (C++, CUDA or OpenCL). Furthermore, the framework has added support for Renderscript targets [34], allowing it to take advantage of heterogeneous hardware in embedded devices. The target language is chosen by the user and the resulting code can be executed on a GPU. This framework was originally built for medical image processing [35]. However, its features allow it to be used for any sort of image processing application.

The embedded DSL provides several interesting mechanisms that allow easier development of image processing pipelines. In addition, the DSL basically consists of C++ template classes provided by the HIPA$^{CC}$ framework. These classes are fully operational, meaning the code can be compiled using standard C++ compilers to run on a CPU. This can be used to compare results between GPU and CPU implementations.

The HIPA$^{CC}$ framework conceptualizes image processing algorithms in a dataflow manner. It views algorithms as a set of kernels that have inputs and outputs. The outputs are stored in a buffer, which in turn can be used as input to another kernel, forming complex pipelines.

Listing 3.1 shows a simple custom filter, using HIPA$^{CC}$. Line 1 sets up the coefficients for this filter using a matrix. This matrix contains the constants that will be used to make the calculations inside the window. The image is loaded on line 10. Lines 12 and 13 set up boundary conditions. Boundary conditions are useful when a kernel tries to access out of bounds pixels. For example, if the operator uses a 5x5 window, it will access out of bounds pixels when it is in position (0,0). The boundary condition allows the values out of bounds to be mapped. In this case, the condition chosen was *BOUNDARY_MIRROR*, which means the values will be mirrored. Finally, the iteration space for the output is created and the filter is built and executed. Executing the filter will call, for each window in the iteration space, the kernel function inside the filter definition, shown in Listing 3.2.

```
 1  const int coefMatrix[5][5] = { { 2, 1, 2, 1, 2 },
 2                                 { 2, 3, 2, 3, 2 },
 3                                 { 7, 2, 6, 2, 7 },
 4                                 { 2, 3, 2, 3, 2 },
 5                                 { 2, 1, 2, 1, 2 } };
 6
 7  Mask <int> mask(coefMatrix);
 8
 9  Image <uchar4> input(width, height);
10  input = image;
11
12  BoundaryCondition <uchar4> boundaryCondition(input, mask, BOUNDARY_MIRROR);
13  Accessor <uchar4> accessor(boundaryCondition);
14
15  Image <uchar4> output(width, height);
16  IterationSpace <uchar4> iterationSpace(output);
17
18  MyFilter myFilter(iterationSpace, accessor, mask);
19
20  myFilter.execute();
```

Listing 3.1: HIPA$^{CC}$ custom filter based on an example taken from Oliver Reiche et al. [36]

```
void kernel() {
  int4 total = { 0, 0, 0, 0 };
  for (int y = 2; y >= -2; --y)
      for (int x = 2; x >= -2; --x)
          total += mask(x, y) * convert_int4(inputAccessor(x, y));
  total = max(total, 0) ;
  total = min(total, 255) ;
  output() = convert_uchar4(total);
}
```

Listing 3.2: HIPA$^{CC}$ Kernel based on an example taken from Oliver Reiche et al. [36]

The kernel function is where the computations happen. For this filter, the values in the window (5x5) are multiplied by their corresponding value in the coefficient matrix. The result is summed. The sum is then normalized to obtain the value for the pixel, which is assigned to the output function in line 8. Note that the traversal of the matrix is entirely hidden from the programmer. All he/she has to do is write the kernel, which is what really matters in the algorithm. The traversal is handled automatically using the framework classes.

Oliver Reiche et al. [36] extended the framework to allow the DSL code to be translated to a form capable of being executed on an FPGA. The extension is focused on generating code for Vivado HLS. It therefore takes the C++ embedded DSL code and generates C++ code that can serve as input to the Vivado HLS suite. Note that the C++ code generated has additional annotations (some of them in the form of pragmas) to allow Vivado HLS to generate better HDL code and thus better FPGA implementations.

The framework is based on the Clang/LLVM compiler infrastructure [37]. The Clang frontend is used to parse the DSL code and generate an Abstract Syntax Tree (AST). This intermediate representation (IR) is then used to generate two kinds of code: the host code and the kernels. The host code is the driver, managing the kernels, launching them and sending them data. The kernel code contains the specified computations to run on the accelerator using the target language chosen (CUDA, OpenCL, Renderscript). The generated AST from the Clang frontend is used to create the necessary adjustments to the target code, so it can be compiled by Vivado HLS.

The host code is generated by obtaining a structural representation of the intended image processing pipeline. This representation is built by traversing the AST looking for buffer allocations, memory transfers and kernel executions. These are identifiable because they use compiler known classes (part of the framework). The resulting representation is a graph that contains nodes for processes (kernel executions) and space (buffers). The graph is traversed backwards in depth first search (DFS) fashion. This way, irrelevant computations are rightfully pruned from the code. The process nodes are translated into kernel executions and the buffers are used to create Vivado HLS stream objects.

Several problems need to be dealt with to generate efficient C++ code. For example, the embedded DSL uses masks to make calculations. When convolutions are calculated using the mask constants, the resulting value range will need to be adjusted, depending on how the mask constants are represented. If the constants are represented as integers, the convolution will require the multiplications and a normalization step at the end. However, if floating point values are used (and all of them add up to 1), then the normalization step at the end is not needed. It turns out that on FPGAs, the former choice has the least impact on resource usage. HIPA$^{CC}$ does not support the use of floating-point values, so the values need to be converted to integers. Note that this conversion requires the additional normalization step at the end.

To generate the device code (kernels), an image processing C++ library is used. Masks are known at compile time (hence the use of constants). This is important to make the necessary transformations to the mask, as described earlier. In addition, the constants can be propagated throughout the code, further improving efficiency. Image dimensions need to be known at compile time as well, to take additional advantage of Vivado HLS and to achieve more efficient FPGA implementations.

Finally, to take further advantage of the Vivado HLS suite, several optimizations are made. Synthesis directives in Vivado can be placed in the code via pragmas or in a separate file. The most common and obvious directives are placed in the code, while the others are put in a separate file to allow users to manipulate them to better tune their designs. Loop counter variables are automatically tuned by Vivado HLS, in order to avoid having to specify the exact necessary bit-width. Note that changing the image dimensions would require changing the bit-width. This way, Vivado HLS does it without help from the user. Vivado HLS uses assertions to infer the required bit-widths. Vector types are also translated to C structures. Note that the motivation behind vector types is that they are usually present in GPU programming models and they are actually a useful way to represent data in image processing algorithms. However, to program FPGAs with Vivado HLS, this choice is not available, so the vector types are translated to structures that can be easily manipulated. Finally, some delays might occur in certain algorithms due to different window sizes in local operators on a pipeline. The solution here is to enforce a delay on the faster operators.

The HIPA$^{CC}$ implementation on an FPGA [36] was evaluated in comparison with implementations using the OpenCV library from Xilinx as well as expert implementations executing on GPUs. The benchmarks used were the Laplacian filter, Harris Corner and the Optical Flow image processing pipelines. The HIPA$^{CC}$ extension was executed on a Xilinx Zynq 7045 FPGA. Other implementations were executed on the Nvidia Tesla K20 GPU and the ARM Mali-T604 embedded GPU. The HIPA$^{CC}$ extension outperforms the Mali by factors of 3 for the Laplacian filter and 19 for the Harris Corner. It outperforms that same GPU for the Optical Flow pipeline by a factor of 456, but this pipeline took abnormally long on the Mali which makes its comparison less relevant. The Tesla outperforms the FPGA by factors of 2 for the Optical Flow and 30 for the Laplacian filter. Throughput shows equivalent values as this metric is highly influenced by performance. Energy efficiency for the FPGA Laplace version is the highest of all the algorithms and it is 19 and 34 times higher than the equivalent Tesla and Mali versions, respectively. Resource usage is better for the HIPA$^{CC}$ extension than the OpenCV equivalents, with 24 times less used LUTs.

Özkan et al. [38] extended the framework even further with the addition of Altera FPGAs through Altera's OpenCL. The approach is somewhat similar to the one used for Vivado HLS, generating a streaming pipeline for execution on the FPGA. Some of the optimizations used are similar to the ones present in the previous extension, although the details of their implementation differ. For example, the bit-width issue mentioned earlier was solved by Vivado HLS itself through the use of arbitrary bit-widths. Although Altera's OpenCL provides a similar mechanism, the OpenCL standard does not. To avoid the loss of portability, this extension uses a bit-wise *AND* to handle the bit-width issue.

Overall, HIPA$^{CC}$ is one of the most complete image processing frameworks out there, as it now provides support for CPUs (using C++), GPUs (using CUDA or OpenCL), Android (using Renderscript), Altera's FPGAs (using Altera's OpenCL) and Xilinx's FPGAs (using Vivado HLS) [39].

### 3.1.2 Halide

Halide [40] is a DSL that can be used to develop image processing pipelines. The Halide compiler can generate code for both CPU execution, as well as for CPU + GPU runs. The language provides several abstractions to aid in the development of image processing algorithms.

Image processing pipelines usually have several stages. Each stage operates on a given input. The input can be an entire image or parts of it. The first stage gets the original image and sends the results of its computation to the next stages and so on until the output is obtained on the last stage. The operations performed by the stages can vary a lot. Some stages might perform stencil computations while others might do resampling or even simple point-wise operations. The key aspect here is that there is a lot of diversity in the type of operations that might be performed. For this reason, choosing the best way to setup the algorithm to obtain the best performance is not a trivial task. This task can be referred to as finding the algorithm's schedule. The schedule specifies the way in which operations are performed and what the storage constraints should be. There are multiple strategies to enhance an algorithm's performance. However, strategies are not always compatible. Many image processing applications try to achieve good performance by making the computations in a parallel fashion. Others might try to achieve better data reuse, because many stages in a pipeline might operate on a window and can thus benefit from reducing redundant computations. These techniques are usually incompatible, because to achieve data reuse, an ordering needs to be enforced, thus sacrificing parallelization. Moreover, the schedule for one of the stages should not be applied to all the other stages, because the proper schedule is stage dependent. That is, a good schedule for one stage might be terrible for another. Additionally, the schedule for a given stage is also dependent on the stages it uses values from. This is an important aspect, because finding the right schedule for the algorithm is not about finding the best schedule for each stage. Ultimately, the key is finding the combination of stage schedules that achieves the best overall algorithm performance. If one considers that some image processing applications have hundreds of stages with a huge amount of different operations, then one can easily understand that scheduling is a very demanding task.

Halide attempts to act on this problem by raising the level of abstraction. Its focus is to decouple algorithms from schedules [41, 42]. The algorithms are the textual representation of the image processing pipelines. This description makes no assumptions about the order in which operations will be performed, nor does it care about storage concerns. The schedule specifies all these characteristics. The DSL uses a function construct to represent a stage in the pipeline. These functions can use other functions so, for the most part, all the stages in the pipeline can be represented in almost the same number of lines. Halide also provides reduction operations. These are represented with an initial value function and a recursive reduction function. They also need a reduction domain, which specifies the order in which the reduction is performed. Reductions can be helpful in operations like histogram equalization, for example.

Listing 3.3 shows a simple example of the Halide syntax. It shows a pipeline with 3 stages. The first one blurs the image over $x$ and the second does the same over $y$. The last stage is the output function and it multiplies the result from the previous stage by a constant. The syntax is fairly straightforward. Note that this specification entails only the computations to be done. It does not say anything regarding the schedule.

```
Func stage1(x, y) = input(x - 2, y) + input(x - 1, y) + input(x, y) + input(x + 1,
    y) + input(x + 2, y)
Func stage2(x, y) = stage1(x, y - 2) + stage1(x, y - 1) + stage1(x, y) + stage1(x,
    y + 1) + stage1(x, y + 2)
Func output(x, y) = stage2(x, y)*10
```

Listing 3.3: Halide example filter based on an example taken from Jonathan Ragan-Kelley et al. [43]

The schedules in the Halide language are defined by telling the compiler when and where each function is computed and where its results are stored. The function traversal can be sequential or parallel. Dimensions with constant size can be unrolled or vectorized as well as reordered, choosing either row or column-major ordering. Dimensions can additionally be split by a factor, thus allowing the grid representing the image being processed to be decomposed as a set of tiles. In the example above, for example, the *output* function dimensions can be split by a factor, effectively tiling that dimension. The same can be done for $x$ and this variable can even be vectorized.

The compilation works by first taking the Halide textual representation of the image processing pipeline and creating a set of loops to execute the algorithm. In this stage, the bounds of the loops and the sizes of the storage buffers are represented by symbolic constants. The second phase is bounds inference. It is here that the bounds for the loops and the buffer sizes are inferred. The third stage looks for sliding window optimizations and storage folding. The focus here is to find better data reuse by looking at which functions use values computed by previous iterations. This avoids redundant computation by sacrificing a bit of parallelism. Storage folding allows the buffers to be shrinked, because many functions compute local operations. That is, many functions use only a subset of the entire buffer (because they might only use the past few scanlines). For this reason, if a function only needs the last 3 scanlines from a buffer, then the buffer can be of size 3, thus reducing the peak memory use and working set size. The fourth stage flattens multi-dimensional loads, stores and allocations into a single dimensional equivalent. Thus, the image is treated as if it were a single line. The fifth stage performs loop unrolling and vectorization according to the schedule. Finally, the Halide IR representation at this point is ran through standard constant folding and dead code elimination passes before being lowered to LLVM IR. Additionally, functions can also be scheduled to run on a GPU. This means that the functions are implemented as kernels that execute on the GPU. If GPU code generation is specified in the schedule, the resulting code is a hybrid implementation (CPU + GPU) of the image processing pipeline.

Previous versions of the Halide DSL required users to specify the schedules by hand. The authors have since developed their own autotuner to aid in this process [43, 44, 45]. The auto tuner applies stochastic search to the schedule search space to find a good schedule. Genetic algorithms are used for the search, with an initial population of 128 schedules. Some of the starting schedules are selected by a weighted coin which chooses, for each function, one of two schedules: a fully parallelized and tiled version or a parallelized over *y* version. Other starting schedules are created applying breadth first schedules to functions (that is, compute and store at the root outermost granularity). The algorithm then iteratively finds new schedules using elitism, crossover, mutations and random individual generation.

The results were measured by developing Halide versions of several algorithms and comparing them to expert implementations in other languages (namely C++). The Xeon W3520 x86 CPU and the NVIDIA Tesla C2070 GPU were used to execute the implementations. The metrics used were speedup and lines of code. GPU versions were also generated for some of the algorithms to see how they perform. The benchmarks used were a blur filter, a camera pipeline, multi-scale interpolation, bilateral grid and local Laplacian filters. The results show that the Halide algorithms get better performance in all the benchmarks while at the same time providing the user with a better experience, because each one requires fewer lines of code. The bilateral grid is by far the most successful in terms of productivity with only 36 lines of code as opposed to the 158 of an expert implementation. The speedup is 4.4 over the expert implementation, for this example, which is the highest of all the speedups. However, all Halide versions have performance gains over the expert implementations. Only one comparison was made to a GPU version, with a speedup of 2.3 for the bilateral grid as well. Local Laplacian filters implemented in Halide for GPU execution have a speedup of 9 over a CPU implementation of the same pipeline (no GPU expert implementation was available). The auto tuner for the examples took between 2 hours to 2 days to find a schedule for each algorithm, which makes it impractical in some real-world applications. However, the performance benefit from using Halide is clear as it generates very fast implementations, which can be useful in some cases (for example, if the input image is of fixed size over time).

Jing Pu et al. [46] developed an extension to Halide to target FPGAs. Their approach uses C target code for Vivado HLS. Their compiler generates a hybrid implementation of the algorithms, with some parts targeting the FPGA and others targeting the CPU. They take advantage of several parts of the original Halide compiler, using the Halide IR to generate a data-flow graph. This graph can then be used to generate hardware implementations of the kernels. For the software section, they rely mostly on the Halide ARM backend, with additional passes to introduce the hardware calls in the correct places. The extension was evaluated in comparison with HIPA$^{CC}$ [36] and an HLS video library. Their results show that their extension to Halide can obtain equivalent results to the other two, saving 6% in BRAM when compared to the HIPA$^{CC}$ for the Harris Corner benchmark. When compared to GPUs, their extension exhibits 1.9x more throughput and 6.1x higher energy efficiency. The results are even better on both metrics when evaluated against the CPU implementations.

### 3.1.3 PolyMage

PolyMage [47] is a DSL used for image processing applications. It is an embedded DSL as it uses Python as host language. The compiler generates C++ code using OpenMP to employ a shared memory model for execution on CPUs. Code is written in the form of image processing stages. Each stage is defined as a function. Function declarations provide the inputs and the definitions give a description of the computation to be made. PolyMage provides several high level image processing related constructs that allow the user to write very complex image processing applications with few lines of code [48].

An example of a kernel specification using the DSL is given in Listing 3.4. Note how the definition of the pipeline is very simple as it uses predefined language constructs like Stencil. Since image processing pipelines have several common operations well defined, these are all given their very own constructs, which reduces development effort. For example, many image processing pipelines use matrices to compute values. These matrices are passed to the Stencil construct in this example, which in turn makes the actual calculations using the parameters it received, hiding implementation details from the user.

```
stage = Function([w, x, y], [intervalWindow, IntervalXRow, IntervalXCol],
                    Float , "stage")
stage.defn = [Stencil(img(w, x, y), 4.5/20,
              [[1], [2], [1]])]
```

Listing 3.4: PolyMage kernel based on an example taken from Nitin Chugh et al. [49]

Nitin Chugh et al. [49] provide an extension to the PolyMage compiler to allow execution on FPGAs. The extension generates C++ code with Vivado HLS directives. It is therefore targeted at the Vivado Design Suite and Xilinx FPGAs. Results show that this extension allows FPGA implementations to reach speedups ranging from 1.05x to 15.60x when compared to CPU optimized implementations. PolyMage FPGA implementations also achieve better performance (1.45x to 1.59x) than Xilinx's predefined OpenCV implementations of two of the algorithms used as benchmarks (Harris Corner Detection and Unsharp Mask).

More recent work by Abhinav Jangda et al. [50, 51] provides an algorithm to improve loop fusion. The case study used to apply the algorithm is PolyMage, although their focus is the image processing domain in general. Therefore, their proposal could be adapted to other DSLs, such as Halide [40]. Their results show improvements against both the original PolyMage and Halide's approaches towards loop fusion.

### 3.1.4 OpenSPL

OpenSPL [52] is a domain-specific language targeting the spatial computing domain. Its focus is providing developers with strong abstractions that allow them to create complex hardware accelerated programs from software defined descriptions, bridging the gap between software and hardware.

The DSL contains a model of Spatial Computing Substrates (SCS), from which different hardware technology can be targeted, from Dataflow Engines from Maxeler to Automata Processor

Engines from Micron or even future Quantum Processing Units [53, 54]. They can thus target FPGAs to allow users to take advantage of their parallel features.

The language extends MaxJ [30], providing additional hardware abstractions more useful to software developers. Computations are done inside kernels. Inputs and outputs are managed using functions, a paradigm already familiar to high level language programmers. A small example of a kernel in OpenSPL is shown in Listing 3.5.

```
class MyKernel extends Kernel {
    MyKernel() {
        SCSVar input = io.input("input", scsFix(12));
        SCSVar output = (input < 25) ? input + 2 : input - 2;
        io.output("output", output, scsFix(13));
    }
}
```

Listing 3.5: OpenSPL kernel based on an example taken from The OpenSPL White Paper [53]

This example shows how branching can be parallelized by mapping the branch to a set of operations that execute in parallel, with their results being inputs to a multiplexer. The branch condition can then be set as the multiplexer selection input to choose the appropriate result, that is, the right branch based on the condition, as shown in Figure 3.1.

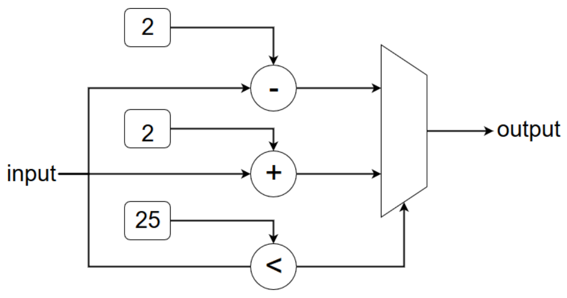


Figure 3.1: OpenSPL branch graph based on an example taken from The OpenSPL White Paper [53]

This shows how the control flow of the kernel shown in Listing 3.5 is mapped into a data flow in Figure 3.1. Moreover, bit widths can be set for each variable used, allowing stronger flexibility.

The OpenSPL compiler generates MaxJ code which can be fed to the MaxCompiler to produce a VHDL description of the application [54]. The bitstream can then be generated for execution on the FPGA.

These are just some of the features available in OpenSPL. The language, while not as high level as general-purpose programming languages like Java or C, still presents a big upgrade from average HDLs such as Verilog or VHDL, which is a benefit for software developers wishing to port their programs to hardware accelerators.

## 3.2 DSLs and frameworks targeting machine learning

This section surveys some related work on DSLs and frameworks targeting machine learning. None of the projects described here has any official support for FPGA execution. Nonetheless, some solutions have been proposed to solve that problem and these are therefore also described throughout each subsection.

### 3.2.1 OptiML

OptiML [55, 56] is a domain-specific language targeting the machine learning domain. It is embedded in Scala and it can be executed on CPUs and Nvidia GPUs. The language uses several machine learning abstractions, such as *Image*, *TrainingSet*, among others. Although these are mapped to either matrices or vectors in the background, they convey certain domain-specific properties that allow the compiler to generate more efficient code. Some of the control structures built into the language are shown in Listing 3.6.

```
// sums
val result = sum(lowerBound, upperBound){ i =>
  <ith value to sum>
}

// creates a new vector
val newVector = (0::end) { i =>
  <ith value of the new vector>
}
```

Listing 3.6: OptiML control structures taken from Arvind K. Sujeeth et al. [55]

These abstractions allow the users to specify their machine learning algorithms in a clear and compact way. Listing 3.7 shows one of the steps in the k-means clustering algorithm.

```
// centroids is a matrix containing the centroids. Each row is a centroid (vector)
// x is a matrix containing the samples. Each row is a sample (vector)
val closestCentroids = (0::m){i =>
  val distances = centroids mapRows { centroid =>
    // distance from the current sample to the current centroid
    ((x(i)-centroid)*(x(i)-centroid)).sum
  }
  distances.minIndex
}
```

Listing 3.7: OptiML k-means centroid calculation taken from Sujeeth et al. [55]

This excerpt calculates the distance between each sample and each centroid and builds a vector containing, for each element, the closest centroid. Vector *closestCentroids* thus contains, for each entry $i$, the index into the centroid matrix that is the closest to sample $i$.

OptiML code is compiled using the Delite Compiler Framework and Runtime [57, 58, 59]. Delite allows the creation of implicitly parallel domain-specific languages that can be targeted to heterogeneous platforms using the Delite runtime. Although Delite users need to make their DSLs embedded in Scala, they can take advantage of certain features built into the framework. One of the key aspects that DSLs take advantage of is the potential for domain-specific optimizations to be performed, leveraging the knowledge the compiler has about the domain. Delite provides users with a modular approach to implementing DSLs that allows users to create domain-specific optimizations for their DSLs by extending certain classes existent in the framework [60]. Moreover, since the framework comes with a variety of predefined compiler optimizations, DSL developers can choose which ones to reuse and which ones to extend.

OptiML code fed to the Delite compiler is used to generate an execution graph: the Delite Execution Graph (DEG). The compiler generates target code for each kernel in the DEG. Delite attempts to generate code in Scala, C++ and CUDA, when possible. The runtime then chooses which version to use during execution. This improves flexibility as it allows the runtime to schedule the kernels in the DEG taking into account dynamic information such as resource availability and input size [61].

While OptiML targets only CPUs and GPUs, George et al. [62] provide an approach to develop FPGA implementations. Their extension targets Xilinx FPGAs using Vivado HLS. In addition to the traditional generic optimizations provided by languages developed in Delite, this extension adds loop unrolling, loop pipelining, as well as the introduction of local buffers and loop sectioning, leveraging domain-specific knowledge.

The extension proposed by George et al. [62] has been evaluated using machine learning benchmarks, such as Nearest Neighbor, Outlier Counter, 1-D Correlation and 1-D Normalization. Versions were implemented in both OptiML and C++ using OpenMP. The former were executed on FPGAs while the latter were exectuted on CPUs. The CPU versions outperform the FPGA versions in performance, although the gap is more evident in the multi-threaded implementations. In terms of energy efficiency, the single-threaded CPU versions spend between 1.8x and 4.4x more energy in fixed point dominated applications than the OptiML FPGA implementations. The multi-threaded CPU versions spend between 1.3x and 3.5x more energy than the OptiML versions executing on FPGAs, on all fixed-point dominated applications.

### 3.2.2 TensorFlow

TensorFlow [63] is an open-source framework for machine learning. It was developed by Google and it is largely based on their experience with the DistBelief system [64]. The framework focuses mostly on neural networks and deep learning models [65], expressing such computations as dataflow graphs. These graphs hold both the computations and the state on which these computations operate on. The models usually perform computations on multi-dimensional arrays called tensors.

TensorFlow can be targeted to a variety of platforms, including CPUs or GPUs [66]. The framework also has support for clusters and mobile devices, allowing applications to be executed on a wide range of computing systems. Despite not having official support for FPGA execution, the framework provides tensor processing units (TPUs) [67] as a possible target platform. TPUs are custom built ASICs targeted at machine learning applications using TensorFlow.

Stable TensorFlow APIs exist for both Python and C. Additionally, support for other languages such as C++, Go, Java, JavaScript and Swift is present, although with no guarantees regarding API backwards compatibility.

Deep neural networks [65] usually contain several layers, each with their own purpose. TensorFlow provides several abstractions to model these networks. An example is shown in Listing 3.8. This Python code creates a network containing 3 layers. The network is supposed to gather images in a 2D format. That is, the input is a 2D array with size 40 in both dimensions. The first layer flattens the array. This transformation basically turns the 2D array into a 1D array (with size $40 \times 40 = 1600$). The next layer computes the rectified linear of the features and it has 100 nodes. The last layer, with 50 nodes, computes the probability scores of each class to be used in classification. The current image will belong to one of the classes based on the probability scores returned by these nodes.

```
// Neural network example
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(40, 40)),
    keras.layers.Dense(100, activation=tf.nn.relu),
    keras.layers.Dense(50, activation=tf.nn.softmax)
])
```

Listing 3.8: TensorFlow neural network Python example taken from [63]

Very little has been done regarding FPGA execution of TensorFlow applications. As mentioned earlier, no official support exists for such a target platform. However, a number of research efforts have focused the FPGA implementation of TensorFlow applications. For example, LeFlow [68] is an open-source tool whose purpose is to synthesize TensorFlow applications for FPGA execution. The tool leverages Google's XLA compiler to generate LLVM code based on a TensorFlow specification. This IR is then subject to several transformations and the resulting IR is used by the HLS tool LegUp [69] to generate a register-transfer level (RTL) implementation. Their tool is capable of targeting TensorFlow applications to FPGAs. However, no experiments were conducted to understand the impact on performance.

### 3.2.3 Theano

Theano [70] is a Python library that provides facilities for mathematical expressions to be computed efficiently. It uses dynamic C code generation to provide fast execution of mathematically intensive operations [71]. It is very popular in the machine learning community as it provides a number of useful abstractions for users in this domain. The library also supports GPU execution using CUDA [72], allowing machine learning applications to be targeted to these devices.

Theano also uses tensors (multi-dimensional arrays) as these are the basis for the computations performed in machine learning. In fact, Theano is based on the popular NumPy library, allowing users already familiar with that library to quickly master Theano concepts. The compilation flow provides several optimizations that allow fast execution times. The computations are also seen as a form of data-flow graphs, although not quite like the ones used by TensorFlow. Some of the optimizations employed by the framework simplify mathematical expressions in order to remove irrelevant computations.

Theano provides support for parallelism in both GPUs and CPUs, taking advantage of multi-core architectures [73]. As mentioned before, the framework uses CUDA to provide GPU execution. In the CPU, OpenMP is used in the C implementations of the main Theano operations, allowing them to be parallelized to increase performance.

A simple 2D convolution in Theano can be computed using the code in Listing 3.9. This particular convolution operates on a 7x7 image and uses a 5x5 window for the computations.

```
// Perform 2D convolution on input image
output = theano.tensor.nnet.conv2d(
    input, filters, input_shape=(1, 1, 7, 7), filter_shape=(1, 1, 5, 5),
    border_mode=(1, 1), subsample=(2, 2))
])
```

Listing 3.9: Theano 2D convolution example taken from [70]

Similarly to TensorFlow, Theano provides a vast documentation with several examples explaining not only the framework but also machine learning in general to some extent. This allows users with no experience in the domain to quickly learn and experiment with the library, mastering machine learning concepts.

Not much has been done regarding FPGA execution for the Theano framework either. However, the FINN framework [74] provides a way for trained binarized neural networks (BNNs) to be mapped to an FPGA using Theano. A Theano trained BNN is input to the framework, which generates a hardware implementation for execution on an FPGA. Much like most experiments done in the FPGA domain, the target environment chosen for this framework is Vivado HLS, allowing it to generate hardware implementations on Xilinx FPGAs.

## 3.3 Other DSLs and frameworks

The previous sections focused on a few DSLs and frameworks targeting both accelerators and the machine learning domain. Additional DSLs and frameworks exist in these and other contexts and some of them are briefly mentioned in this section.

While TensorFlow [63] and Theano [70] are very popular frameworks in the machine learning domain, other frameworks have been developed with somewhat the same purpose. Caffe [75] is a framework focusing on deep learning, targeting CPUs and GPUs (using CUDA). Roberto DiCecco et al. [76] provide an approach to target Caffe convolutional neural networks (CNNs) to an FPGA. PyTorch [77] is a Python machine learning library focusing on both tensor operations and deep learning. Like many others, this library provides both CPU and GPU target code generation. Tensors have also been used in other contexts. One example is CFDlang [78], a DSL targeting fluid dynamics making extensive use of tensors. In fact, these constructs have become so popular that meta-languages focusing on tensors themselves have been proposed as well [79].

Other domains have also seen extensive use of DSLs. PPME [80, 81] is an environment for the development of particle mesh simulations. It makes extensive use of equations and mathematical formalisms, providing users with a more mathematically friendly interface. Darkroom [82] is a DSL focusing image processing, targeting both FPGAs and CPUs. Much like the DSLs described throughout Section 3.1, it views image processing algorithms as complex pipelines, containing several stages, each with its own computations. RIPL [83] is another DSL focusing image processing for FPGA targets. It is motivated by the requirements of the domain, viewing programs as dataflow graphs as well. SPIRAL [84, 85] allows automatic generation of hardware implementations of digital signal processing algorithms, such as the Fast Fourier transform. DIF [86] is a DSL focusing digital signal processing as well. Similarly to the previous DSLs showcased throughout this chapter, it views programs as dataflow graphs. A number of DSLs and tools have also been developed to automate the solution of differential equations [87, 88].

A popular trend among many of the languages shown in this chapter is the decomposition of programs as a graph, where nodes usually represent computations. Many DSLs have been developed with this paradigm in mind, but targeted at other domains. Acme [89] is a DSL focusing the software architecture domain. It decomposes systems as a set of components that are connected to each other. LEDA [90] is another DSL targeting this domain as well, with an emphasis on dynamic architectures. xBreeze/ADL [91] is a XML-based DSL also centered on software architecture and specification. The concept of streaming has also seen extensive work, with DSLs proposed to handle stream processing applications, some of which even target FPGAs [92, 93].

Chamberlain et al. conducted a study using crowdsourcing techniques to understand preferences in language design [94]. Their approach considers two styles for streaming languages: functional and literal. These styles (or parts of them) can be identified in all the languages depicted in this chapter. Moreover, the DSL developed in this dissertation is slightly inspired in the literal style presented there.

## 3.4 Summary

The previous sections reviewed related work on both DSLs and frameworks, not only in the machine learning domain, but in other fields as well. Tables 3.1 and 3.2 overview the main DSLs studied throughout this chapter.

Table 3.1: DSLs properties overview

| DSL | Type | Programming Model | Target Languages/Tools | Domain | Other Computing Devices | Optimizations |
|---|---|---|---|---|---|---|
| HIPA$^{CC}$ [33] | Textual; C++ Embedded | Dataflow; Imperative; Object Oriented | Altera OpenCL; Vivado HLS C++ | Image Processing | CPU/GPU | Mask transformations; Constant propagation; Loop counter tuning; Vector type translation |
| Halide [40] | Textual | Dataflow | Vivado HLS C | Image Processing | CPU/GPU | Loop bounds inference; Buffer size inference; Sliding window opt.; Storage folding; Flattening; Vectorization; Constant-folding; Dead-code elimination |
| OpenSPL [52] | Textual | Dataflow | MaxJ | Spatial Computing | N/A | N/A |
| PolyMage [47] | Textual; Python Embedded | Dataflow | Vivado HLS C++ | Image Processing | CPU | Buffer allocation opt.; Fusion; Tiling; Data forwarding; Mask transformations; Arbitrary data types |
| OptiML [55] | Textual; Scala Embedded | Imperative | Vivado HLS C++ | Machine Learning | CPU/GPU | Common subexpression elimination; Constant propagation; Dead code elimination; Code motion; Loop unrolling; Loop pipelining; Loop sectioning; Local buffers |

It is fairly evident that image processing is by far one of the most popular domains when it comes to DSLs for FPGA execution. Several DSLs exist in this domain and as such, users have a wide range to choose from when they decide to target image processing applications to these devices. Machine learning and data analytics are clearly missing more work regarding FPGA execution. As mentioned throughout Section 3.2, machine learning DSLs and frameworks lack official support for FPGA execution. Although several authors have proposed approaches to extend both OptiML [55] and frameworks like TensorFlow [63] and Theano [70] for FPGA execution, these are not official and can thus quickly get outdated. Therefore, no support is expected when targeting these devices.

Another problem that is evident when one looks at the current state of the art in machine learning frameworks is their focus on neural networks and deep learning. Machine learning is a very vast domain. Including support for every single machine learning algorithm in a DSL or framework is not a trivial task. Therefore, it is expected that many of the frameworks and DSLs identified focus on a small subset of machine learning algorithms and models. However, the development of new frameworks and DSLs focusing on other sub-domains other than neural networks and deep learning would provide great contributions to the machine learning domain in general.

Given the current state of the art in data analytics, it is clear that the development of a DSL targeting this domain for FPGA execution would allow developers to more quickly build efficient

Table 3.2: DSLs evaluation overview

| DSL | Benchmarks | Evaluation |
|---|---|---|
| HIPA$^{CC}$ [33] | Laplacian filter;<br>Harris corner;<br>Optical flow | Performance;<br>Throughput;<br>Energy;<br>Power;<br>Resource usage (Slices, LUTs, FFs, DSPs, BRAM) |
| Halide [40] | Blur filter;<br>Camera pipeline;<br>Multi-scale interpolation;<br>Bilateral grid;<br>Local Laplacian filters | Performance;<br>Productivity |
| OpenSPL [52] | N/A | N/A |
| PolyMage [47] | Unsharp Mask;<br>Harris Corner Detection;<br>Optical Flow | Performance;<br>Throughput;<br>Resource usage (Slices, LUTs, FFs, DSPs, BRAM) |
| OptiML [55] | Nearest neighbor;<br>Outlier counter;<br>1-D correlation;<br>1-D normalization | Performance;<br>Resource usage;<br>Energy |

data analytics systems for execution on these devices. Allowing these applications to take advantage of the FPGA architecture would ultimately contribute to both the FPGA and data analytics communities.

# Chapter 4

# A new DSL for Data Analytics

This chapter presents the developed DSL. A brief description of the DSL engineering process is first given, explaining the though process behind some of its key features. These features are then further explained using some representative examples to better demonstrate the most relevant use cases. Details on the compilation process are also explored, with a special focus on targeting FPGA-based systems.

## 4.1 DSL Engineering

This section details some of the main requirements for the new DSL for data analytics. Some key issues are also discussed, to better understand the design decisions taken in the development of the DSL.

### 4.1.1 The Data Analytics Domain

As mentioned earlier, data analytics systems often make extensive use of machine learning algorithms. These algorithms are usually characterized by deep pipelines containing several stages. Each stage in the pipeline usually does some computation on the input values it receives. The output it produces is then communicated to another stage in the pipeline. The main idea is shown in Figure 4.1.

This configuration is very similar to the one used in other application domains, such as image processing. The fact that the algorithms can be decomposed in several stages is what makes the entire domain appealing for FPGA execution after all, as this setting allows for massive amounts of parallelism to be exploited. Moreover, the stages themselves often perform operations that can be subject to parallelization, thus improving overall performance.

For these reasons, mapping this visual representation to a textual one within the DSL seems to be the right approach to the problem at hand. Note that in addition to being able to map the visual image of the pipeline to a textual representation, this configuration also allows the code to be more structured and readable. Moreover, it promotes code reuse because stages can be used by different pipelines and pipelines themselves could be integrated in a bigger pipeline.

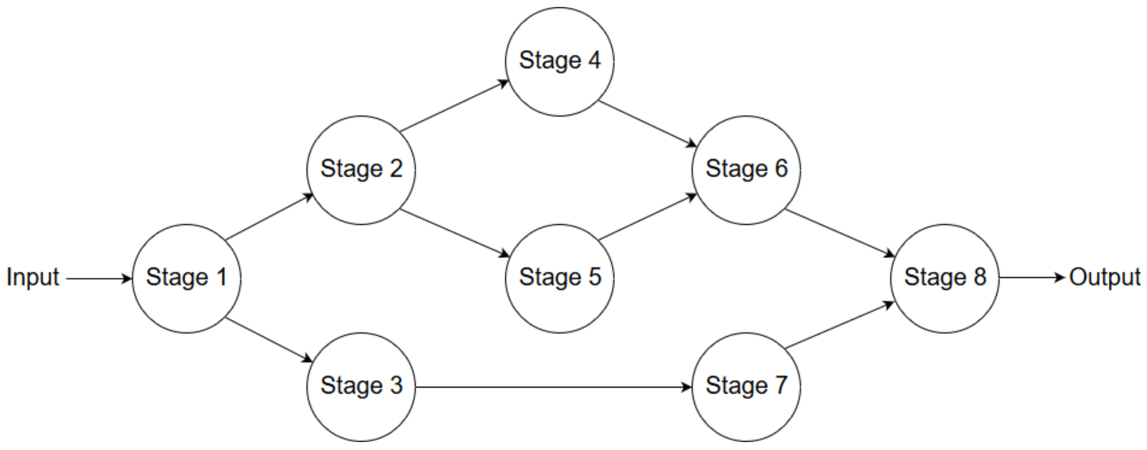


Figure 4.1: Machine learning pipeline

### 4.1.2 Targeting FPGAs

Targeting FPGAs can be done using either a hardware description language (HDL) or a high-level language using high-level synthesis (HLS). One of the most popular HLS environments is Vivado HLS [15]. This environment can produce FPGA bitstreams from C or C++ code. It is a tool developed by Xilinx so it targets this vendor's FPGAs. Therefore, targeting C code, synthesizable by Vivado HLS is an interesting approach. This process was illustrated earlier in Chapter 1 in Figure 1.2.

One important issue to take into consideration is whether or not the DSL should be coupled with the FPGA target. That is, should the DSL be aware of the FPGA target? Taking into account the fact that the DSL could ideally be extended to other platforms, making it aware of the FPGA target seems unfitting. As such, the DSL could focus solely on CPUs, leaving any non conventional targets to non conventional compilers. Therefore, instead of enforcing standard DSL compilers to target FPGAs and interpret FPGA specific code, the DSL shall provide a mechanism to allow it to be extended by other compilers. This shall be done using a concept already existent in many other languages: pragmas. The language shall recognize any pragmas. However, it does not need to interpret any of them. This is left to each specific compiler. Section 4.4 dives further into this detail, showcasing a pragma that declares a region to be targeted to an FPGA.

### 4.1.3 The DSL requirements

The main DSL and compiler requirements are shown in Table 4.1.

One of the main goals with the DSL is to allow the graphical representation of a machine learning pipeline to be textually specified using stages. The stages and interfaces between them shall be decoupled, because this allows greater independence. Stages can thus be used multiple times by the same pipeline. The order in which stages and interfaces are specified shall not matter either. Therefore, the compiler shall infer the machine learning pipeline graph from the stages and interfaces specified. The DSL shall also provide mechanisms to read data from a file. Another key

Table 4.1: The main DSL and compiler requirements

| Requirement | Description |
|---|---|
| **Stage composition** | The code shall be comprised of stages, each with their own inputs, outputs and behaviors (the computation to be done by the stage) |
| **Stages and interfaces decoupling** | The stages and the interfaces shall be decoupled, allowing the same stage to be used multiple times in the pipeline (by different interfaces) |
| **No stage ordering** | The compiler shall not require the stages to be specified in any specific order. The computation pipeline shall be automatically inferred |
| **Input/Output** | Inputs and outputs for the first and last stages shall be simple and easy to express, allowing users to send data to and out of the pipeline in a simplified manner. Files shall be a valid choice for input and output |
| **Built-ins** | The DSL shall provide a set of built-ins for commons mathematical and machine learning related operations |
| **Overlapping mechanism** | The stages shall allow for an overlapping factor to be set |
| **Heterogeneous execution** | The DSL compiler shall allow hybrid implementations, using the CPU to manage the application control flow. The code in the pipelines shall be targeted for FPGA execution |
| **Pragmas** | The DSL shall provide pragmas, in order to allow specific stages to be targeted to an FPGA |
| **C integration** | The DSL shall allow C implementations to be integrated in the pipeline as an additional stage, allowing users to specify hybrid pipelines containing DSL and hand-tuned C code |

aspect to be focused is the machine learning domain, so built-ins shall be provided for common mathematical and machine learning operations. This eases the development effort. Additionally, the fact that the compiler is aware of these constructs allows better optimizations to be applied. Moreover, the stages shall allow for an overlapping factor to be set, thus making it possible to overlap values from consecutive iterations of the same stage. This is very important because many machine learning pipelines often use this feature. Heterogeneous execution is another relevant topic for the compiler. Even though the DSL is not tied to the FPGA target, the compiler developed during this dissertation shall allow for FPGAs to be used, either standalone or in a hybrid scenario (using a CPU as well). Finally, an interesting feature that could be explored as well is the integration of C code into a DSL written program.

## 4.2 The new DSL

Taking the requirements noted in the previous section into account, this section presents the DSL. Appendix A.1 contains the full grammar for the developed DSL and Listing 4.1 shows a simplified version.

```
<Program> -> <Decl>+
<Decl> -> <ConstDecl> | <AliasDecl> | <StageDecl> | <InterfaceDecl>
<StageDecl> -> 'stage' <Identifier> ('with' 'overlapping' <OverlappingExpr>)? '{'
               <StageProp>+ '}'
<OverlappingExpr> -> <Identifier> | <DecimalLiteral>
<StageProp> -> <Input> | <Output> | <Behavior> | <Setup>
<InterfaceDecl> -> 'interface' '{' <From> <To> '}' ';'?
<Input> -> 'input' (<InputDecl> | '{' <InputDecl>+ '}' ';'?)
<InputDecl> -> 'fillable'? <VariableDecl> ';'
<Behavior> -> 'behavior' (<Statement> | '{' <Statement>+ '}' ';'?)
<Output> -> 'output' (<OutputDecl> | '{' <OutputDecl>+ '}' ';'?)
<OutputDecl> -> <VariableDecl> ';'
<Setup> -> 'setup' (<Statement> | '{' <Statement>+ '}' ';'?)
<From> -> 'from' (<FromFile> | <FromStage>) ';'
<FromFile> -> 'file' (<RootType>)? <StringLiteral> ('repeat' | 'header')?
<FromStage> -> 'stage'? <Identifier> ('.' <Identifier>)? ('with' 'overlapping'
               <OverlappingExpr>)?
```

Listing 4.1: DSL grammar excerpt

### 4.2.1 Declarations

A program written in the DSL is a sequence of declarations. Declarations can be of four types: *constant*, *alias*, *stage* or *interface*. Constant or alias declarations are fairly straightforward. They are used to define constant variables and type aliases respectively (they are somewhat equivalent to C's *define* directive and *typedefs*). An example of *const* and *alias* declarations is shown in Listing 4.2.

```
// Defines a constant with value 18.
const N = 18;

// Defines an alias for the double type.
alias double myType;
```

Listing 4.2: The *const* and *alias* keywords

A stage in a program is essentially a processing node, as depicted in Section 4.1. The stage contains a set of inputs, outputs and a behavior. A stage is required to have at least one input and one output. The behavior operates on the inputs and produces the outputs. Within the stage's behavior, variables may be declared and manipulated. The DSL allows statements within the *behavior* block, similar to those used in other languages such as C. Appendix A.1 provides more details about the statements supported. A simple stage is shown in Listing 4.3. This stage is identified by the name *mean* and it contains two inputs: *a* and *b*. The output of the stage is an integer named *result*. The *behavior* tag identifies the computation done in the stage. In this case, the *result* output is assigned the value of summing inputs *a* and *b*.

```
// Computes a + b.
stage mean {
    input {
        int a;
        int b;
    }
    behavior result = a + b;
    output int result;
}
```

Listing 4.3: A simple *stage*

Having stages allows one to describe what sort of computations are intended for a stage in a given machine learning pipeline. However, the pipeline itself must also be specified. That is, the pipeline graph must be described by the user. This is where interface declarations come in. An *interface* declaration provides information about one or more edges in the pipeline graph. An interface contains a source node and one or more destination nodes. In this manner, a user specifies the edges in the pipeline using interface declarations. Listing 4.4 shows a simple interface declaration that connects stage *a* to stage *b*.

```
// Describes an interface from stage a to stage b.
interface {
    from a;
    to b;
}
```

Listing 4.4: A simple *interface*

### 4.2.2 Input and Output

One thing that was left unexplained in the previous subsection is how the program gathers inputs and what happens to the outputs. Machine learning algorithms often use datasets as input to the pipeline. These datasets are usually read sequentially and the values obtained are fed to the pipeline for classification. The developed DSL provides a way to read from a file directly into a stage. This is done using an interface declaration, as shown in Listing 4.5. This example shows several more details regarding interface declarations. The first one is the *file* keyword. This is used here after *from*, meaning that the program will be reading from the file named "sample.dat". Note that machine learning algorithms operate on one instance at a time. That is, the entire dataset will not be fed to the pipeline all at once. As such, in many cases, the dataset will be sent to the pipeline one line at a time. In this case, the values in the 4th, 5th and 6th columns of the file are being sent to three different stages. Listing 4.6 shows the *mean* stage, which computes the mean of 3 values.

```
// Sends the contents in the 4th, 5th and 6th columns of file "sample.dat" to the
// "mean", "variance" and "signalMagnitude" stages.
interface {
    from file "sample.dat";
    to {
        4, 5, 6 -> mean;
        4, 5, 6 -> variance.points;
        4, 5, 6 -> signalMagnitude;
    }
}
```

Listing 4.5: An *interface* from a file

```
stage mean {
    input double[3] points;
    behavior result = avg(points);
    output double result;
}
```

Listing 4.6: The *mean* stage

Notice how the *mean* stage has only one input: an array of size 3. The interface declaration shown in Listing 4.5 specifies that columns 4, 5 and 6 of the file shall be input to the *mean* stage. Note, however, that this does not send values from all three columns to the exact same instance of that stage. The code in Listing 4.6 shows the declaration of the stage. In reality, the stage can be used more than once in the same pipeline. As such, an instance of the stage is required for each use. To make it simpler, the DSL does not require the user to handle instances explicitly. Instead, they are implicitly created by the compiler. In this case, when the interface states that the 4th, 5th and 6th column are sent to the *mean* stage, these are actually sent to different instances of that stage. This means that this interface creates three new instances of the *mean* stage, one for each column.

A key aspect of this stage is also how the input array is filled. When an interface declaration is specified involving an array input (such as in the *mean* stage), the array is not actually filled all at once. Instead, the array is filled one row at a time. In this case, a row is one value only (because this is a 1D array). This means that for this stage in particular, the input array requires 3 iterations to be filled. That is, three values are read from the file and fed to the stage. Therefore, the interface provided in Listing 4.5 creates 3 new instances of the *mean* stage and each one computes the mean of 3 values in that column. Note that the pipeline runs indefinitely (or at least until there is no more input), so the stage is called for every three values read.

Note that the interface in Listing 4.5 does not specify what input in the *mean* stage should the values in the file be fed to. This is because the stage only has one input and as such, the compiler looks for the first valid one it finds. Notice how the same interface states that the same values are also sent to the *variance* stage. However, in this case, the input to send the values to is specified (*points*). This is because that stage has more than one input, as shown in Listing 4.7. The *variance* stage computes the variance of a set of points, taking into account their mean, which is also an input to the stage. The stage's behavior uses tensor operations and mathematical primitives to express the variance computation. These operations and primitives are explained later.

```
stage variance {
    input {
        double[N] points;
        double mean;
    }
    behavior result = sum((points - mean)^2)/N;
    output double result;
}
```

Listing 4.7: The *variance* stage

Another important detail about inputs is present in the *signalMagnitude* stage (see Listing 4.8). This stage has three array inputs. This raises an interesting point. What should the compiler do when it finds the interface shown earlier in Listing 4.5? That interface specified that columns 4, 5 and 6 go to the *signalMagnitude* stage. For the *mean* stage, this introduced three new instances of that stage. However, for this one, only one instance of *signalMagnitude* is created. This is because the input to send the values to within the stage was not specified (as was done for the *variance* stage). As such, the compiler tries to find a valid input for each column. It maps the 4th column to the *x* input, the 5th column to the *y* input and the 6th column to the *z* input. Therefore, only one instance is created, which is exactly what was intended. If, on the other hand, each column required its own instance, then all that is needed is for the exact input to be specified in the interface, just like it is done for the *variance* stage.

```
stage signalMagnitude {
    input {
        double[3] x;
        double[3] y;
        double[3] z;
    }
    behavior result = sqrt(x^2 + y^2 + z^2);
    output double[3] result;
}
```

Listing 4.8: The *signalMagnitude* stage

Note that this mechanism is the same used for the *mean* stage. The difference is that in the *mean* stage, the compiler needs to create new instances because the instances already created are full (i.e., have no inputs available).

A final relevant aspect regarding input and output is how one can print the results of the program to check for errors. This is done using the *stdout* keyword, as shown in Listing 4.9. The current version of the DSL does not support outputting to files, but future work could add this feature, using a similar approach to how files are read (see Listing 4.5).

```
interface {
    from classifier;
    to stdout;
}
```

Listing 4.9: Interfacing to *stdout*

### 4.2.3 Mathematical primitives and operators

The stages shown in the previous subsection revealed a few more features in the DSL: the *avg*, *sum* and *sqrt* mathematical primitives, and the ˆ operator. The DSL provides a few built-in primitives and operators to handle common operations such as the square root. These are depicted in Table 4.2.

Table 4.2: DSL mathematical primitives and operators

| Primitive/Operator | Description |
|---|---|
| **sum**(*tensor*) | Computes the sum of the values in the *tensor*. The *tensor* can be of any rank |
| **avg**(*tensor*) | Computes the average of the values in the *tensor*. The *tensor* can be of any rank |
| **sqrt**(*tensor*) | Computes the square root of the values in the *tensor*. The tensor can be of any rank. The result is a tensor where each element is the square root of the corresponding element in the input tensor |
| *a*ˆ*b* | Computes the power involving base tensor *a* and scalar *b*. The base tensor can be of any rank. The result is a tensor where each element is the result of computing the power of the corresponding element in the original base tensor and the exponent |

Another interesting aspect taken from these stages is the way tensor operations are performed. Notice how the + operator is used in the *signalMagnitude* stage. Inputs *x*, *y* and *z* are all arrays. This means that *x*ˆ2 computes a new tensor where each value is squared. The same is done for the two remaining inputs. The resulting tensors are then added, which corresponds to creating a new tensor where each value is the sum of the values in the corresponding positions of the original tensors. Thus, the stage computes the square root of an array of size 3, which itself outputs a new array, where each element is the square root of the element in the corresponding position of the original array.

This type of computations is usually done on the target programming language (e.g., C) using loops. This notation allows the loops to be completely left out. Note, however, that the compiler can still generate loops to execute operations. They are just invisible to the user in the source code of the DSL, which emphasizes readability.

### 4.2.4 Types

Something that has not been touched upon yet is the type system. The DSL only allows the use of primitive types. These types are fully depicted in Appendix A.2. Table 4.3 shows the most relevant ones with a simplified description.

Table 4.3: DSL main types

| Type | Description |
|---|---|
| **boolean** | Boolean type. 1 byte long |
| **char** | Character type. 1 byte long |
| **short** | Signed short type. 2 bytes long |
| **int** | Signed integer type. 4 bytes long |
| **long** | Signed long type. 8 bytes long |
| **float** | Floating point type. 4 bytes long |
| **double** | Floating point type. 8 bytes long |

Most of the types in the DSL are the ones present in C. Boolean is the only one that is not. The *boolean* type is mapped by the compiler to a *char* in the target language.

### 4.2.5 Machine learning primitives

Other than the fact that the DSL allows one to build machine learning pipelines in a structured manner, the features shown so far have not been too attached to the domain itself. The DSL contains a set of additional features that are oriented towards machine learning based solutions.

The first one is the overlapping mechanism. As mentioned earlier, machine learning pipelines are often composed as a set of stages. These pipelines often require an initial phase where features are extracted. The operations to perform in either of these phases often involve the use of several values from an input set. For example, one might want to read the input set and compute the averages of every group of 3 values. However, one might wonder what 3 values to choose from. Should the computation always be done with 3 new values or should some of them be overlapped with the following iterations? This detail can be configured in the DSL using the *with overlapping O* notation, as shown in Listing 4.10. Whenever this stage is used by an interface, only 1 new value is required per iteration (except for the first one, of course). As such, the last 2 values in the *points* array will be used in the following iteration of the stage. The default overlapping factor for any stage is 0.

```
const N = 3;
const O = 2;

// overlapping = 2
stage mean with overlapping O {
    input double[N] points;
    behavior result = avg(points);
    output double result;
}
```

Listing 4.10: The *mean* stage with overlapping

Sometimes, the overlapping mechanism needs to be set specifically for an instance instead of a stage in general. Listing 4.11 shows how the overlapping mechanism can be overridden for one instance. This example uses the *with overlapping 0* notation after a stage reference inside the interface to specify that the stage shall use a different overlapping factor (in this case, 0).

```
interface {
    from signalMagnitude;
    to {
        mean with overlapping 0;
        variance with overlapping 0;
    }
}
```

Listing 4.11: Overriding the overlapping factor of the *mean* and *variance* stages for a specific instance

After feature extraction, a classifier is usually called to classify a feature vector. This process often requires the use of a training set. The developed DSL provides builtin primitives to handle these issues as well. Table 4.4 shows the machine learning primitives currently supported by the DSL.

Table 4.4: DSL machine learning primitives

| **Primitive/Operator** | **Description** |
|---|---|
| **readTrainingSet**(*file*,*n*,*w*) | Reads the training set present in *file*. The number of samples is *n* and the number of columns is *w*. The last column is the class ID |
| **readTrainingSetToHeader**(*file*,*n*,*w*) | Similar to *readTrainingSet* but the file is read during compilation and a static array is generated in a separate header with its contents |
| **knn**(*fv*,*ts*,*k*,*nc*) | Executes the k-nearest neighbors algorithm. The parameters are the feature vector *fv*, the training set *ts*, the value of *k* and the number of classes *nc* |

To read a training set, the *readTrainingSet* primitive can be used. This primitive returns a 2D array with the contents of a file given in a parameter (the file contains the training set). The classifier can then be called with the training set and the feature vector in order to find out the class of the given instance. An example using the *knn* primitive is shown in Listing 4.12. This code introduces several new details. First off, the setup block of a stage is being used. The setup block in a stage is similar to the behavior block. The only difference is that the setup block is only executed once. Hence, it is adequate for initialization code. This is important in this use case, because we only need to read the training set once (in the beginning). Thus, we put the *readTrainingSet* primitive inside the setup block in order for the the training set to be read once before any executions begin.

Note that the *readTrainingSet* primitive returns a 2D array with the feature vectors for several instances in addition to their IDs. As such, the length of the array is actually $NUM\_FEATURES + 1$. Additional work could focus on separating this information in two different arrays: one for the training set and another one for the IDs.

```
// Constants used.
const K = 3;
const NUM_FEATURES = 12;
const NUM_CLASSES = 25;
const NUM_TRAINING_SAMPLES = 529;

// The stage with the classifier.
stage classifier {
    setup double[NUM_TRAINING_SAMPLES][NUM_FEATURES + 1] trainingSet =
        readTrainingSet("trainingSet.dat", NUM_TRAINING_SAMPLES, NUM_FEATURES + 1);
    input fillable double[NUM_FEATURES] featureVector;
    behavior result = knn(featureVector, trainingSet, K, NUM_CLASSES);
    output double result;
}
```

Listing 4.12: The *classifier* stage using the k-nearest neighbors (k-NN) algorithm

Another key aspect of the DSL introduced in Listing 4.12 is the *fillable* keyword. Recall that any input array is filled one row at a time. That is, when an array is provided as input to a stage, the compiler will assume that the array will require *N* iterations to be filled and be ready for use by the stage. This is the desired behavior for many stages, like the *mean* or *signalMagnitude* stages shown earlier. However, sometimes an input array should be filled with values coming from different stages. In the case of the *mean* stage, the input array was always filled with values coming from the same place (a column in the file). However, in this case, the *featureVector* input is supposed to hold several values coming from different stages. For instance, the first element of the array might be the result of calling stage *mean* with a given column. The second element might correspond to the variance. Clearly, every element is unique in the sense that they could, in fact, all have their own separate variables. However, such an approach would not be very flexible, which is why a feature vector is useful. After all, the order in which the elements appear is not really relevant, as long as it remains the same throughout the execution of the pipeline (and the training set feature vectors use that same order). With the *fillable* keyword, multiple stages can send their outputs to the same input (the one marked *fillable*). This way, the *featureVector* array is filled with the right contents. Note that the same could be achieved by considering the *featureVector* as a 2D array with only one row and *NUM_FEATURES* columns. This would present the same behavior. However, it could be very confusing. As such, the *fillable* keyword was introduced.

One final detail worth mentioning about the classifier stage is the classifier itself, in this case, k-NN. The k-NN algorithm can be called using the builtin primitive *knn*. This primitive takes four arguments: the feature vector, the training set, the value of *k* and the number of classes. The primitive's return value is a class ID.

### 4.2.6 Scope

One issue that has not been discussed yet is how the scope mechanism works within the DSL. Every declaration written on the top level of the program is global. This means that any stages, constants or aliases are accessible anywhere in the program after they have been declared. In addition, any of these are also accessible within any stage. This means that a constant can actually be defined at the very bottom of the program, as long as it is only used within a stage. Moreover, the order in which the stages and interfaces are defined is irrelevant. A stage can be defined after an interface that uses it. Additionally, interfaces do not need to follow any specific order. The pipeline graph is automatically inferred based on the interfaces and stages provided.

### 4.2.7 Templates

Sometimes the type of the variables being used significantly impacts performance. Therefore, it is important to have the ability within the DSL to choose the right types for each operation. For simple operations such as addition, everything is implied from the types of the variables involved (or by casts). For other operations, such as those requiring a read from a file, it is not as simple. For these cases, the DSL allows users to select the types for each primitive being used. Examples are shown in Listings 4.13 and 4.14. The default type used when reading from a file is *double*. However, Listing 4.13 shows how an *int* can be used instead. If the same behavior is needed when reading a training set, the *readTrainingSet* family of primitives also allows type templating, as shown in line 3 of Listing 4.14.

```
interface {
    from file<int> "sample.dat";
    to {
        4, 5, 6 -> mean;
    }
}
```

Listing 4.13: Setting a type for the *file* constructor

```
stage classifier {
    setup int[NUM_TRAINING_SAMPLES][NUM_FEATURES + 1] trainingSet =
        readTrainingSet<int>("trainingSet.dat", NUM_TRAINING_SAMPLES,
                                    NUM_FEATURES + 1);
    input fillable int[NUM_FEATURES] featureVector;
    behavior result = knn(featureVector, trainingSet, K, NUM_CLASSES);
    output int result;
}
```

Listing 4.14: Setting a type for the *readTrainingSet* primitive

### 4.2.8 Using header files

A perhaps unconventional feature present in the DSL is the ability to read files during compilation and write them to a static array in a separate header file to be used by the target code. This is supported both when reading from a file within an interface declaration or when a training set is read. Both cases are shown in Listings 4.15 and 4.16. The first example uses the *header* keyword to specify that the file should be read during compilation and written to a static array. The second example uses the *readTrainingSetToHeader* primitive to achieve the same behavior when reading from a training set.

```
interface {
    from file "sample.dat" header; // Notice the "header" keyword.
    to {
        4, 5, 6 -> mean;
        4, 5, 6 -> variance.points;
        4, 5, 6 -> signalMagnitude;
    }
}
```

Listing 4.15: The *header* keyword when reading from a file

```
stage classifier {
    setup double[NUM_TRAINING_SAMPLES][NUM_FEATURES + 1] trainingSet =
        readTrainingSetToHeader("trainingSet.dat", NUM_TRAINING_SAMPLES,
                                    NUM_FEATURES + 1);
    input fillable double[NUM_FEATURES] featureVector;
    behavior result = knn(featureVector, trainingSet, K, NUM_CLASSES);
    output double result;
}
```

Listing 4.16: Reading a training set to a header file

Note that having the contents of the files in a static array prior to execution is useful for prototyping. It is also worth noting that in a real scenario, the values would probably be coming from sensors and the pipeline would run in an infinite loop. Such a scenario is not supported but could be explored in future work.

## 4.3 Compilation Flow

This section focuses on the compilation flow of the DSL, explaining what each phase in the compiler does and how target code is generated. The DSL compiler follows the process shown in Figure 4.2.

The following subsections go through each phase, explaining what is done at every step. The compilation flow follows a structure familiar with other compilers, including lexical, syntax and semantic analysis phases before code generation [95]. An optimization step is also included, although this only applies to code targeted to FPGAs.

### 4.3.1 Lexical Analysis

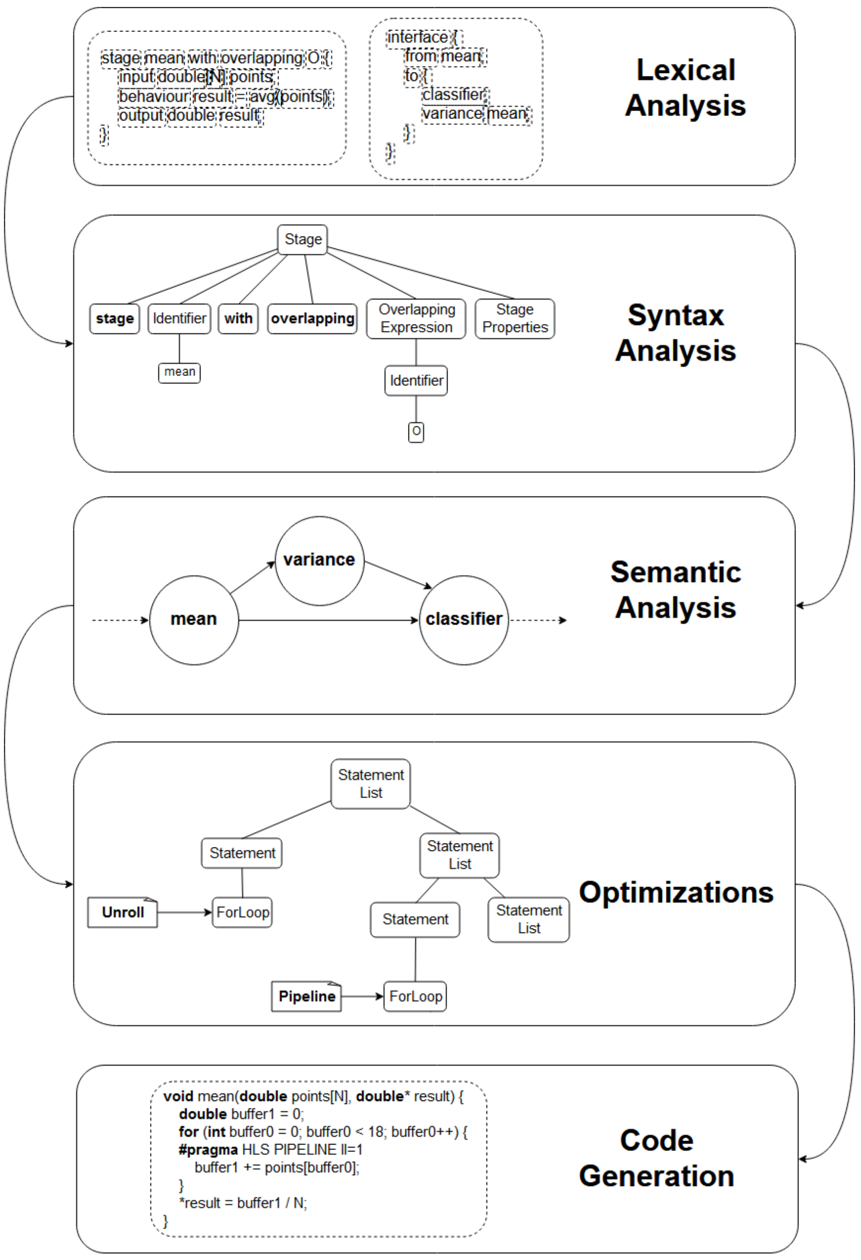
As with most compilers, the first phase is the lexical analysis phase. This phase reads the input source code and outputs a chain of tokens. This phase makes heavy use of regular expressions to recognize the lexems of the language. The regular expressions used by the DSL can be consulted on Appendix A.1. The output of this phase of the compiler is a stream of tokens that are input to the syntax analysis stage.

### 4.3.2 Syntax Analysis

The tokens provided by the previous phase are used during the syntax analysis phase to parse the program. This phase interprets the code. To do so, a grammar is used. The grammar for the language can be consulted on Appendix A.1. A syntax tree is then formed and walked to build an intermediate representation (IR) of the program. This IR contains a set of data structures that represent the program pipeline. As such, it includes stages, interfaces and so on. This phase also makes heavy use of a symbol table to generate symbol table entries for each declaration. As such, certain semantic checks can be verified during this phase. For example, any top level declaration that uses an undefined variable will throw an error here.

### 4.3.3 Semantic Analysis

The semantic analysis phase takes care of all remaining semantic checks. Note that not everything can be checked by walking the parse tree once. This phase performs type checking, scope checking and other domain-specific validations such as verifying that each stage contains at least one input and one output. This phase makes extensive use of the intermediate representation built earlier. In fact, that representation is enhanced, allowing the program to finally be complete.



```
stage mean with overlapping O {
    input double[N] points;
    behaviour result = avg(points);
    output double result;
}
```

```
interface {
    from mean;
    to {
        classifier;
        variance mean;
    }
}
```

```
void mean(double points[N], double* result) {
    double buffer1 = 0;
    for (int buffer0 = 0; buffer0 < 18; buffer0++) {
    #pragma HLS PIPELINE II=1
        buffer1 += points[buffer0];
    }
    *result = buffer1 / N;
}
```

Figure 4.2: DSL compilation flow

Note that certain aspects within the DSL require the entire program to be parsed before the full IR can be built. For example, to build the pipeline graph, the compiler needs to first read all the interfaces and store them. It then needs to infer the graph based on the nodes they contain. This process can be very tricky so an example can be of use here. Consider the code shown in Listing 4.17. The code contains 5 interfaces. Note that the interfaces are in a user friendly order to improve readability, but they could be placed in any order. For this reason, the compiler needs to first read all the interfaces and only then can it start inferring the pipeline graph. The first step in inferring the right graph is finding source nodes, that is, nodes that are not a destination in any interface. Once these have been found, the graph can be easily inferred using an iterative approach. The resulting graph for the code below is shown in Figure 4.3.

```
interface {
    from file "sample.dat";
    to {
        4, 5, 6 -> mean;
        4, 5, 6 -> variance.points;
        4, 5, 6 -> signalMagnitude;
    }
}

interface {
    from mean;
    to {
        classifier;
        variance.mean;
        standardDeviation.mean;
    }
}

interface {
    from variance;
    to {
        classifier;
        standardDeviation.variance;
    }
}

interface {
    from standardDeviation;
    to classifier;
}

interface {
    from signalMagnitude;
    to {
        mean with overlapping 0;
        variance with overlapping 0;
    }
}

interface {
    from classifier;
    to stdout;
}
```

Listing 4.17: A set of interfaces within a program

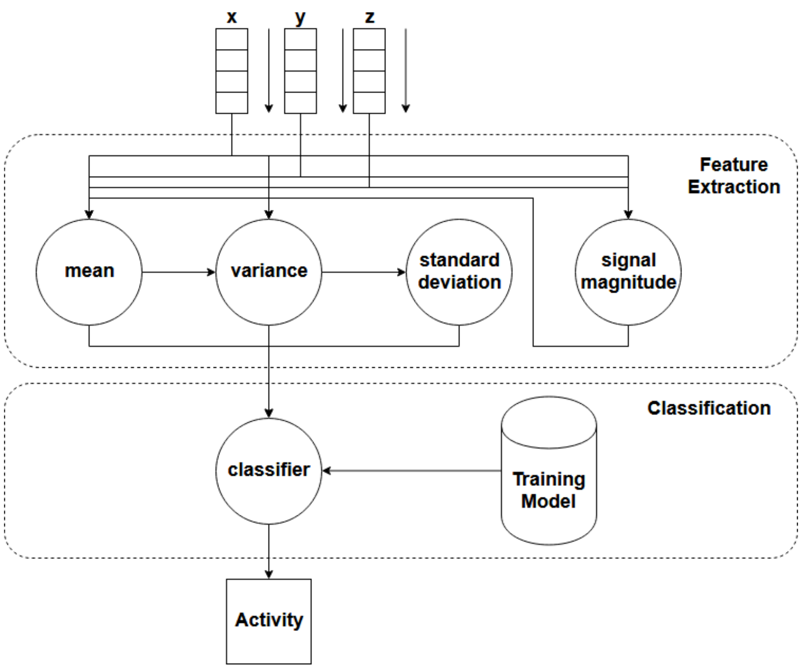


Figure 4.3: The block diagram resulting from the interfaces shown in Listing 4.17

There is one aspect that makes this process more complex. Note that Figure 4.3 shows only one node for each stage, even though they are called upon several times within the pipeline. The figure clearly illustrates how the pipeline works. However, the fact that stage instances are not explicitly handled by the user within the DSL makes the job of the compiler more demanding. In addition to inferring the graph, the compiler needs to be able to set up the instances automatically. This is done by reserving each input in a stage. That is, when the compiler is trying to process a connection between two nodes, it looks for a stage instance with the requested input free. If it does not find one, a new instance is created. If a stage has multiple inputs and no input was specified, then the compiler tries to match any input. Note that the behavior for *fillable* inputs is different, as these allow multiple stages to fill them in the same iteration. As such, when a *fillable* input is found, a new instance is not created, unless the input is already full (every element filled).

### 4.3.4 Code Optimization

This phase takes care of optimizations to the intermediate code. The IR is thoroughly examined in order to find potential for optimizations. This phase of the compiler is optional and can be triggered using a compiler flag. This dissertation focused FPGAs, so optimizations were only made when the target is an FPGA. These optimizations are explained in Section 4.4. Note that the FPGA target option is specific of the compiler developed. As explained in Section 4.1, the DSL is completely unaware of the FPGA target.

### 4.3.5 Code Generation

This is the last phase of the compiler. This phase gathers the intermediate representation built in the previous phases (with or without optimizations) and generates code for it. The code generated in this case is organized in two sections. One of the sections is the pipeline code while the other is the stages code. Each stage is a C function. These functions are called by the pipeline function. The pipeline function contains the main loop. This loop goes through the input samples (from a file, for example) in an iterative fashion and calls the stages in the pipeline whenever ready.

Listing 4.18 shows the generated code for the *variance* stage, which is a function containing three loops, because this stage uses the *sum* primitive and operates on tensors. Note that additional buffers are used to store intermediate results. In this case, the first loop calculates the $points - mean$ sub-expression used in the stage's *behavior*. This result is then squared using the second loop, which uses the *math.h* function *pow* (if floats were used, the *powf* function would be used instead). Note that to optimize the code, the *pow* function can be substituted here by a mere multiplication, because the exponent is known at compile time to be 2. This transformation is applied when the compiler is called with optimizations, as shown in Section 4.4.3. The third loop sums the values in the resulting tensor. Finally, the result of this sum is divided by $N$.

```c
void variance(double points[N], double* mean, double* result) {
  double buffer7[18];
  for (int buffer6 = 0; buffer6 < 18; buffer6++) {
    buffer7[buffer6] = points[buffer6] - *mean;
  }
  double buffer5[18];
  for (int buffer4 = 0; buffer4 < 18; buffer4++) {
    buffer5[buffer4] = pow((buffer7)[buffer4], 2);
  }
  double buffer3 = 0;
  for (int buffer2 = 0; buffer2 < 18; buffer2++) {
    buffer3 += buffer5[buffer2];
  }
  *result = buffer3 / N;
}
```

Listing 4.18: The generated code for the *variance* stage

Listing 4.19 displays a portion of the main pipeline loop. This loop starts by picking up values coming from the file. The file is read before hand, so the contents of the file are in a buffer. This buffer is used to copy the values in the requested columns to the buffers to be used by the stages. The loop also uses additional buffers to know when to call each stage. For example, the *mean* stage is only called here when a specific counter hits 18. This is because the size of the input array to that stage is 18. As such, the stage can only be called when the inputs are filled. After the stage call, additional buffers are filled with the stage's outputs. These buffers are used by any subsequent stages this stage connects to. Finally, the overlapping regions need to be updated. This is done using a loop that copies overlapping values to the correct positions. The stage counter is updated according to the overlapping factor.

```
// ...
for (int buffer74 = 0; buffer74 < 100000 && buffer74 < buffer29; buffer74++) {
    buffer27[buffer31] = buffer28[buffer74][4];
    buffer31 = buffer31 + 1;
    buffer41[buffer52] = buffer28[buffer74][4];
    buffer52 = buffer52 + 1;
    buffer33[buffer35] = buffer28[buffer74][5];
    buffer35 = buffer35 + 1;
    buffer43[buffer56] = buffer28[buffer74][5];
    buffer56 = buffer56 + 1;
    buffer37[buffer39] = buffer28[buffer74][6];
    buffer39 = buffer39 + 1;
    buffer45[buffer60] = buffer28[buffer74][6];
    buffer60 = buffer60 + 1;
    if (buffer31 == 18) {
      mean(buffer27, &buffer30);
      buffer47[buffer49] = buffer30;
      buffer49 = buffer49 + 1;
      buffer51 = buffer30;
      buffer54 = buffer54 + 1;
      for (int buffer32 = 0; buffer32 < 0; buffer32++) {
        buffer27[buffer32] = buffer27[buffer32 + 18 - 0];
      }
      buffer31 = buffer31 - 18 + 0;
    }
    // ...
}
```

Listing 4.19: An excerpt of the generated main loop code

## 4.4 Targeting FPGAs

Targeting FPGAs requires two steps. The first one is including a way to choose the target to generate code for. The second is making sure the generated code is valid for that target. An optional third step can be used to optimize the code to the target. In this dissertation, optimization is only done for FPGA targets, as mentioned in Section 4.3.4.

### 4.4.1 Using pragmas

As mentioned earlier, the DSL is completely unaware of the FPGA target. However, the DSL supports the use of pragmas. Pragmas are compiler directives that can be used to extend functionality within the language by certain compilers. In this case, pragmas are useful to choose a target for each stage. Note that a compiler flag can also be used to set the target to generate code for. However, such an approach targets all the code to the same architecture, which is often not what is intended. As such, this compiler interprets *TARGET* pragmas, as shown in Listing 4.20. In this example, the *variance* stage uses the default target which is the CPU. The *standardDeviation* stage explicitly chooses the CPU as target, while the *signalMagnitude* stage chooses the FPGA.

```
// Uses the default target (CPU).
stage variance with overlapping 0 {
    input {
        double[N] points;
        double mean;
    }
    behavior result = sum((points - mean)^2)/N;
    output double result;
}

#pragma TARGET CPU
stage standardDeviation {
    input double variance;
    behavior result = sqrt(variance);
    output double result;
}

#pragma TARGET FPGA
stage signalMagnitude with overlapping 0 {
    input {
        double[N] x;
        double[N] y;
        double[N] z;
    }
    behavior result = sqrt(x^2 + y^2 + z^2);
    output double[N] result;
}
```

Listing 4.20: Use cases of the TARGET pragma

### 4.4.2 Generating valid code

Generating valid code is a vital issue when targeting non conventional architectures such as FPGAs. For example, if no memory optimizations are used for an input or output array in a stage, then the *memcpy* C function cannot use that array. The compiler uses *memcpy* for array assignments when targeting the CPU. However, for FPGA targets, such an approach might not always be possible, so adjustments must be made.

It should be noted that targeting FPGAs usually involves adding additional code to improve performance. In the context of Vivado HLS [15], this means adding HLS pragmas. Note, however, that the code should be executable even without the pragmas and if the compiler is called without optimizations, then no pragmas are generated.

### 4.4.3 Optimizations

Vivado HLS provides several performance optimization directives in the form of HLS pragmas [96]. Some of the most common ones are shown in Table 4.5. These directives must be carefully selected by the compiler when the FPGA target is chosen for a given stage. Note that assigning pragmas to the code might increase the program performance but it might also significantly impact the FPGA resource usage. As such, different time/space trade-offs can be achieved depending on what pragmas are used.

Table 4.5: Common directives used in Vivado HLS

| Directive | Description |
|---|---|
| Pipeline | Pipelines a region. The region can be a loop or a function. The initiation interval is reduced, allowing subsequent iterations to begin execution before previous ones finish. A target initiation interval can be requested.<br>Example:<br>`for (int i = 0; i < 10; i++) {`<br>`#pragma HLS PIPELINE II=1` |
| Unrolling | Unrolls a loop. Unrolling can be done partially (by specifying a factor) or fully. Unroll decreases the number of iterations in the loop, merging work from multiple iterations in a single one.<br>Example:<br>`for (int i = 0; i < 10; i++) {`<br>`#pragma HLS UNROLL FACTOR=2` |
| Inline | Inlines a function. This means the code of the function substitutes the function call, therefore minimizing function call overhead.<br>Example:<br>`void func() {`<br>`#pragma HLS INLINE` |
| Dataflow | Allows data to flow through a region, exploring task-level parallelism. Each task (loop or function) can execute at the same time, and data is sent from one task to the other.<br>Example:<br>`void func() {`<br>`#pragma HLS DATAFLOW` |

Two of the most popular directives used are pipelining and unrolling. Both directives can be applied to loops and pipelining can also be applied to functions. When applied to loops, both optimizations allow the parallel execution of multiple iterations, although using different approaches.

Loop pipelining [95] minimizes the initiation interval of a loop, allowing a new iteration to start executing before the previous one has finished. This is done by dividing the instructions inside the loop, in order to make a pipeline. The data is sent through the pipeline and each node executes some operation. This effectively exploits paralellism as several operations can be executing at the same time. In the context of Vivado HLS [15], a target initiation interval can be set for this optimization. However, sometimes it is impossible to achieve certain target intervals. As such, Vivado HLS can sometimes fail to meet the target initiation interval. The default target initiation interval is 1.

Loop unrolling [95] minimizes the number of iterations in a loop, by merging multiple iterations into a single one. If an unrolling factor is used, the loop is partially unrolled, meaning the number of iterations drops by the factor chosen. If no factor is used, the number of iterations is dropped to 0. That is, the loop is fully unrolled. Loop unrolling can sometimes lead to better results than loop pipelining. However, it is usually more costly, leading to higher resource usage.

Table 4.6 overviews the optimization strategy used by the compiler. The compiler uses both loop unrolling and loop pipelining. Although they can both be used together, this use of the two can lead to worse results, as shown in Chapter 5. As such, this compiler chooses to either pipeline or unroll a loop. The target initiation interval set when loop pipelining is used is 1. When loops are unrolled, no factor is used, meaning the loop is fully unrolled. Two levels of optimization were considered. The first one (O1) focuses on space in addition to speed, while the second one (O2) favours speed. For the first optimization level, loops with less than 5 iterations are unrolled, leaving all the other loops pipelined. For the second optimization level, loops up to 50 iterations are unrolled. All other loops are pipelined. When loop nests are present, the outer loop is usually the one optimized, unless the inner loops include many iterations, in which case the inner loops are optimized instead. Note that pipelining an outer loop automatically unrolls inner loops. Therefore, care must be taken when optimizing loop nests.

In addition to these optimizations, the compiler also optimizes the *math.h* function *pow*, by replacing it with multiplications when the exponent is known during compilation. This is done for exponents below 10. Additional work could focus on finding the optimal value at which the use of *pow* becomes more beneficial.

It should be noted that the compiler walks, for the most part, through the final IR instead of the original one. That is, the code optimizer mostly handles IR nodes that are not specific of the source language (in this case, the DSL). For example, if a stage uses a mathematical primitive that requires loops, the loops will be examined instead of the primitive itself. This is motivated by the fact that future work on optimizations could focus on code restructuring in addition to pragma generation. As such, walking through a fine grained IR makes code manipulation easier. With this approach, if loop merging was introduced to the compiler, this optimization could be performed with less effort. Note that this is also possible using the original primitives, but from a compiler

Table 4.6: Compiler optimization strategy

| Statement | Optimization Level | Condition | Optimization Applied |
|---|---|---|---|
| For Loop | O1 | Inner loops combined number of iterations < 5; Number of iterations < 5 | Fully unroll the loop |
| | | Inner loops combined number of iterations < 5; Number of iterations >= 5 | Pipeline the loop (II = 1) |
| | | Inner loops combined number of iterations >= 5 | Optimize inner loops |
| | O2 | Inner loops combined number of iterations < 50; Number of iterations < 50 | Fully unroll the loop |
| | | Inner loops combined number of iterations < 50; Number of iterations >= 50 | Pipeline the loop (II = 1) |
| | | Inner loops combined number of iterations >= 50 | Optimize inner loops |
| Pow | O1 and O2 | Exponent < 10 | Replace the *pow* function with a multiplication |

design perspective, it's easier to work with the loops, since these are all instances of the same node (the loop). If the primitives themselves were walked by the compiler, adding a new primitive would require much more work, because a new node in the IR would need to be handled by the optimizer. Note that this approach is not always beneficial, however, since this is a DSL after all. Certain domain-specific optimizations require knowledge about the actual primitives, so in those cases, the code optimizer would use the primitive node of the IR, instead of any sub-nodes.

Listing 4.21 shows an excerpt of the generated code for the *classifier* stage when optimization level *O1* is used (recall that *O1* optimizes for speed and space). The code contains two loops that are generated by the *knn* primitive. The first one is a loop nest with two inner loops. This loop computes the distances between the feature vectors in the training set and the input feature vector. It also continuously updates the $K$ best points array (in this case, *buffer24*). The inner loops combined number of iterations is 16 ($13+K$), since $K=3$. Therefore, the outer loop is not optimized, and the inner loops are optimized instead. The first inner loop computes the distance between two points. It is pipelined since it has 16 iterations. The second loop updates the array with the $K$ best points (*buffer24*), by finding the worst points and replacing them with better ones. This is done taking into account the distance calculated in the previous loop. The loop is unrolled since it only has 3 iterations. Recall that loops with less than 5 iterations are unrolled. The second loop (after the loop nest) finds the class most training samples belong to. This loop contains no inner loops and has $NUM\_CLASSES + 1$ iterations. Since $NUM\_CLASSES = 25$ and $25 \geq 5$, this loop is also pipelined.

```
// ...
for (int buffer30 = 0; buffer30 < 529; buffer30++) {
  double buffer31 = 0;
  for (int buffer32 = 0; buffer32 < 13; buffer32++) {
  #pragma HLS PIPELINE II=1
    int buffer33 = 0;
    if (buffer32 != 12) {
      buffer31 += ((featureVector[buffer32 - buffer33] - trainingSet[buffer30][
          buffer32 - buffer33]) * (featureVector[buffer32 - buffer33] - trainingSet
          [buffer30][buffer32 - buffer33]));
    } else {
      buffer33 = 1;
    }
  }
  double buffer34 = 0;
  int buffer35 = 0;
  for (int buffer36 = 0; buffer36 < K; buffer36++) {
  #pragma HLS UNROLL
    if (buffer24[buffer36] > buffer34) {
      buffer34 = buffer24[buffer36];
      buffer35 = buffer36;
    }
  }
  if (buffer31 < buffer34) {
    buffer24[buffer35] = buffer31;
    buffer25[buffer35] = buffer30;
  }
}
// ...
for (int buffer65 = 0; buffer65 < NUM_CLASSES + 1; buffer65++) {
#pragma HLS PIPELINE II=1
  if (buffer52[buffer65] > buffer53) {
    buffer53 = buffer65;
    buffer54 = buffer65;
  }
}
// ...
```

Listing 4.21: An excerpt of the generated code for the *classifier* stage using optimization level *O1*. For this code, $K = 3$ and $NUM\_CLASSES = 25$

Listing 4.22 shows an excerpt of the generated code for the *classifier* stage using the *O2* optimization level (recall that *O2* optimizes for speed). This time, since the inner loops' combined number of iterations is 16 and $16 < 50$, the outer loop is optimized. Note that the outer loop has 529 iterations. Therefore, the loop is pipelined (since $529 \geq 50$). Finally, the second loop is unrolled since $26 < 50$.

```
// ...
for (int buffer30 = 0; buffer30 < 529; buffer30++) {
#pragma HLS PIPELINE II=1
  double buffer31 = 0;
  for (int buffer32 = 0; buffer32 < 13; buffer32++) {
    int buffer33 = 0;
    if (buffer32 != 12) {
      buffer31 += ((featureVector[buffer32 - buffer33] - trainingSet[buffer30][
          buffer32 - buffer33]) * (featureVector[buffer32 - buffer33] - trainingSet
          [buffer30][buffer32 - buffer33]));
    } else {
      buffer33 = 1;
    }
  }
  double buffer34 = 0;
  int buffer35 = 0;
  for (int buffer36 = 0; buffer36 < K; buffer36++) {
    if (buffer24[buffer36] > buffer34) {
      buffer34 = buffer24[buffer36];
      buffer35 = buffer36;
    }
  }
  if (buffer31 < buffer34) {
    buffer24[buffer35] = buffer31;
    buffer25[buffer35] = buffer30;
  }
}
// ...
for (int buffer65 = 0; buffer65 < NUM_CLASSES + 1; buffer65++) {
#pragma HLS UNROLL
  if (buffer52[buffer65] > buffer53) {
    buffer53 = buffer65;
    buffer54 = buffer65;
  }
}
// ...
```

Listing 4.22: An excerpt of the generated code for the *classifier* stage using optimization level *O2*. For this code, $K = 3$ and $NUM\_CLASSES = 25$

Note that additional optimizations could be used, but the strategy chosen here is motivated by the experiments conducted in Chapter 5. The results shown there explore more optimizations, including ones involving code restructuring. Code restructuring, however, is much more complex and requires a deeper understanding of the code. Introducing this in the compiler is therefore left as future work.

## 4.5 Implementation Details

This section discusses some details regarding the work developed with a focus on the implementation. The design of the DSL has required an engineering step explained throughout Section 4.1. To develop the compiler and the DSL, several guidelines were taken to make sure the code was ready to be extended. Several design patterns [97] were used, readability was emphasized and the compiler was extensively tested. To make things simpler, a parser generator was used. Multiple technologies were surveyed [98, 99, 100, 101], in order to select one that makes development easier. In the end, the technology chosen was ANTLR [98]. With ANTLR, the lexer and the parser are automatically built from a grammar (the grammar shown on Appendix A.1). For details into the compiler usage, consult Appendix A.3.

The current version of the compiler contains over 75,000 lines of code (including tests and Javadoc annotations). About 5,000 of those lines are from the lexer and parser generated by ANTLR. There are 217 classes used by the compiler (not counting the classes used by test cases). There are also 1279 unit tests and 17 integration tests, making the code coverage of the entire compiler close to 100% for most packages.

## 4.6 Summary

This chapter detailed the developed work of this dissertation. The DSL was introduced using several examples to explain the most relevant features. The compiler was also discussed, with a special focus on FPGA target optimizations.

It is clear that the DSL has many useful features to work on data analytics solutions. Its focus is data analytics programs using machine learning algorithms and, as such, the presence of machine learning primitives makes it a lot easier for developers to program their applications. The use of several mathematical primitives that can operate on any tensor also improves the development process, allowing complex operations to be expressed in a single line, as opposed to a few loops. In addition, readability is also favoured, with the decomposition of stages being a decent bridge between the textual and the graphical representations of the program pipeline.

In the end, the DSL has potential for improvements in several departments. The most relevant one at the moment would be the pragma generation strategy, as the one currently implemented is very simple. Models could be designed to have a better pragma selection policy, improving application performance. Moreover, the optimization phase could take other factors into account, such as communication. Code restructuring is also an effective mechanism to improve performance so it would also be a good fit for future work on the compiler. Furthermore, the compiler could also leverage the knowledge it has on the domain to perform additional domain-specific optimizations. For example, machine learning algorithms are often evaluated according to their accuracy. The accuracy, however, does not always need to be the absolute best. Small changes to a program can lead to a small decrease in classifier accuracy, but a strong improvement in performance. This type of domain specific analysis could be integrated into the compiler as well.

The DSL itself could also be the focus of several additional improvements. The only machine learning algorithm currently supported is k-NN [12] but new ones could be added. As machine learning is a very vast growing field, adding new primitives may bring new challenges on its own, as many algorithms are very different from each other.

Verification is another part that could be further addressed. Verification of machine learning applications raises interesting questions and it can be difficult to debug faulty programs. As such, adding program verification could be beneficial towards developers. For example, one common mistake that can happen using the DSL in its current version is using an overlapping factor in the wrong place, leading to inconsistent throughputs between stages. This kind of mistake can be difficult to trace, so making the compiler aware of such situations would be valuable for the developer.

The type system could be further explored as well. The addition of more fixed-point data types could be an interesting avenue to pursue. Furthermore, the introduction of custom types to handle domain-specific constructs could be a valuable addition to the language as well. For example, the current machine learning primitives operate on tensors only, but future work could focus on custom data types resembling popular machine learning structures such as a training set or a feature vector.

One aspect that was depicted in the initial requirements of the DSL was the integration of C code. Although there was no time to develop this feature, it would certainly be a very interesting one to explore, as it has potential to make the DSL even more powerful and expressive.

Finally, as mentioned earlier, in a real scenario, the values to be used by the pipeline could be coming from sensors, for example, instead of files. The files use case is needed for evaluation but the DSL could be extended with features to support real embedded systems situations.

# Chapter 5

# Evaluation

This chapter shows the experimental results obtained in order to evaluate the DSL and the compiler. First, an overview of the experimental setup is given, providing some context into the benchmarks used as well as the CPU and FPGA specs. The DSL is then evaluated in terms of productivity and performance. The performance results also dive into possible improvements that could be made to the DSL. The chapter ends with a discussion on lessons learned based on the results obtained.

## 5.1 Experimental Setup

To evaluate the DSL, the pipeline shown in Figure 4.3 was used. This pipeline represents a human activity recognition (HAR) system [102] consisting of five stages. Four of them are part of feature extraction, computing the mean, variance and standard deviation of three coordinates ($x$, $y$ and $z$). These operations are also performed on the result obtained from the *signalMagnitude* stage. The classifier (k-NN) is then called, with $k = 3$ and a training set containing 529 samples, each belonging to one of 25 classes. The input data corresponds to $x$, $y$ and $z$ values from an accelerometer (data from the PAMAP2 Physical Activity Monitoring dataset [103] was used). The number of features used in this example is 12, the size of the window during feature extraction is 18 and the overlapping factor is 2.

The following sections address both productivity and performance for the sample pipeline just described. It is worth to note that the generated code was manually modified to create certain versions, in order to understand the impact of different optimizations.

For the performance evaluation, a Zedboard is used, which has a Xilinx Zynq-7000 AP SoC XC7Z020-1CLG484C. The CPU experiments are done using the 667 MHz Dual-core ARM Cortex A9 CPU present in the Zedboard. The FPGA clock frequency used for the experiments is 100 MHz. For the high-level synthesis estimates, Vivado HLS 2018.3 [15] is used. The mixed CPU-FPGA versions are compiled using SDSoC 2018.3 [21]. For the CPU sections, the GCC compiler with the *-O3* compilation flag is used, in order to achieve the best performance.

Table 5.1: DSL and generated C code statistics for the HAR example (using floats). A simplified version of the HAR system (without the *signalMagnitude* stage) is also included

| Version | Source | | | | Code Generated | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | LOC | Stages | Interfaces | Arrays | LOC | Functions | Loops | Arrays | Pragmas |
| HAR CPU | 95 | 5 | 6 | 8 | 380 | 7 | 29 | 27 | 0 |
| HAR FPGA | 95 | 5 | 6 | 8 | 382 | 7 | 30 | 27 | 0 |
| HAR FPGA (w/ *-O2*) | 95 | 5 | 6 | 8 | 397 | 7 | 30 | 27 | 15 |
| Simplified HAR CPU | 76 | 4 | 5 | 4 | 277 | 6 | 19 | 20 | 0 |
| Simplified HAR FPGA | 76 | 4 | 5 | 4 | 277 | 6 | 19 | 20 | 0 |
| Simplified HAR FPGA (w/ *-O2*) | 76 | 4 | 5 | 4 | 286 | 6 | 19 | 20 | 9 |

## 5.2 Productivity

Productivity is usually difficult to quantify. Therefore, it is not easy to evaluate the DSL in this regard. However, Table 5.1 shows some statistics gathered for the source and generated code, using multiple versions. The simplified HAR version shown here is the original one without the *signalMagnitude* stage.

The DSL can clearly use less lines of code than C. The number of stages is almost the same as the number of functions. As mentioned in Section 4.3.5, each stage is generated as a function in the target code, leading to this similarity. Note that the source code has no loops (they are not even supported in the DSL's current version). However, the primitives used by the example make extensive use of loops, leading to a significant loop count. In addition, the main loop also requires additional loops to handle aspects such as the overlapping regions updates.

The generated code uses a lot more arrays than the source code. This is due to the pipeline. The source code versions only use arrays within stages. The generated versions, on the other hand, require additional arrays to send data from one stage to the other. Moreover, the primitives used by the DSL sometimes require additional buffers to store intermediate results.

The simplified HAR versions present a significant decrease in lines of code, loops and arrays. This happens because the *signalMagnitude* stage is removed in this version. The *signalMagnitude* stage uses a number of mathematical primitives and tensor operations, which leads to an increase in the number of loops to compute the intermediate results (from 19 to 30), and the number of arrays to store them (from 20 to 27).

Finally, the optimized versions add pragmas to all the loops inside stages (note that the stages were all targeted to an FPGA in those versions), in order to improve performance, which increases the lines of code slightly. As mentioned in Section 4.4.3, code restructuring was not used when optimizing the code. If such an approach was taken, the generated code would suffer a lot more changes, as loops would get merged and intermediate buffers would be eliminated.

It is worth noting that the compiler relies on a rather generic approach to generate the C code. As such, certain sections of the code could be very different if the versions were manually developed from scratch. For example, the number of loops could naturally be reduced since many of them can be merged. This would make the code easier to understand. In addition, it would reduce the number of arrays used, since less intermediate results would need to be stored. Note, however,

Table 5.2: Contributions of each stage to the entire pipeline (using floats). These values are obtained executing the stages on the CPU using timers

| Stage | Notes | Clock cycles | Contribution (%) |
|---|---|---|---|
| Mean | - | 294714 | 0.99 |
| Variance | - | 894750 | 3.01 |
| Standard Deviation | - | 120418 | 0.41 |
| Signal Magnitude | - | 587206 | 1.98 |
| Classifier | - | 27157360 | 91.38 |
| Feature Extraction | Grouped | 1591686 | 5.40 |
| | Grouped. Merging sub-functions | 836060 | 2.90 |
| Global | - | 29719078 | - |
| | Grouping feature extraction | 29488614 | - |
| | Grouping feature extraction. Merging sub-functions | 28801918 | - |

that this sort of approach doesn't necessarily lead to better results in terms of performance, as shown in Section 5.3. Another aspect that would most likely be very different in a manually developed version is the number of buffers used to send values from one stage to another. The current version of the compiler creates a buffer for every connection. This leads to multiple buffers holding the same values. A developer would most likely create a single buffer in these situations and use it on several stages. Although some of these changes can effectively be made by a compiler using optimizations, a manual version would most likely still use less loops and buffers than a generated one. This advantage could, however, come at a cost in flexibility, because if the C code is too attached to the use case in place, changing the tiniest detail might require a lot more effort than when using the DSL.

## 5.3 Performance

Performance on an FPGA is affected by the accelerator code and the communication between the accelerator and other components (e.g., CPU). The accelerator code can be improved using high-level synthesis (HLS) directives, as mentioned earlier. The communication section requires communication directives. As mentioned earlier, the compiler currently only adds HLS directives to the code. However, the results shown in this section explore both accelerator code and communication, showing what speedups can be obtained when the right optimizations are used. Future work could be done on the pragma generation strategy to include communication directives, based on the results obtained here.

### 5.3.1 Stage contributions

Before targeting a stage to a hardware accelerator, it is worth exploring the potential for performance improvements for each of the stages in the pipeline. Table 5.2 shows the contributions of each stage to the pipeline when executing on the CPU. For completeness, additional versions were created where the feature extraction phase is grouped in a single stage. One of those versions also merges the sub-functions used by the feature extraction phase.

The results demonstrate that the classifier is by far the most important stage, taking 91.38% of the execution time of the entire pipeline. Even when all the feature extraction functions are

Table 5.3: High level synthesis performance estimates for each stage with floats. Best speedups are shown in bold

| Stage | Notes | Clock Cycles | Clock Period (ns) | Execution Time (ms) | Speedup |
|---|---|---|---|---|---|
| Mean | Without directives | 142 | 7.26 | 1.03 | - |
| | Pipeline loops (O1) | 108 | 7.26 | 0.78 | **1.31** |
| | Unroll loops (O2) | 107 | 9.38 | 1.00 | 1.03 |
| Variance | Without directives | 684 | 9.39 | 6.42 | - |
| | Pipeline loops | 174 | 9.39 | 1.63 | 3.93 |
| | Pipeline loops. Replace pow (O1) | 157 | 8.02 | 1.26 | 5.10 |
| | Pipeline loops. Unroll loops (f = 2). Replace pow | 122 | 14.51 | 1.77 | 3.63 |
| | Unroll loops. Replace pow (O2) | 116 | 9.49 | 1.10 | 5.83 |
| | Merge loops. Pipeline resulting loop. Replace pow | 117 | 7.26 | 0.85 | **7.56** |
| Signal Magnitude | Without directives | 1789 | 9.39 | 16.79 | - |
| | Pipeline loops. Replace pow (O1) | 174 | 8.13 | 1.41 | 11.87 |
| | Unroll loops. Replace pow (O2) | 35 | 9.20 | 0.32 | **52.18** |
| | Dataflow function. Unroll loops. Replace pow | 44 | 8.13 | 0.36 | 46.95 |
| | Merge loops. Pipeline resulting loop. Replace pow | 46 | 8.13 | 0.37 | 44.91 |
| Classifier | Without directives | 238839 | 9.63 | 2300.97 | - |
| | Pipeline loops. Replace pow | 37638 | 15.21 | 572.47 | 4.02 |
| | Unroll loops. Replace pow | 38674 | 9.63 | 372.59 | 6.18 |
| | Pipeline nested loop. Unroll other loops. Replace pow (O2) | 3298 | 9.63 | 31.77 | **71.42** |
| | O1 | 37101 | 15.21 | 564.31 | 4.08 |
| Feature Extraction Grouped | Without directives | 3745 | 9.39 | 35.15 | - |
| | Best combination of functions. Replace pow | 693 | 9.80 | 6.79 | 5.17 |
| | Merging functions. Replace pow | 284 | 9.90 | 2.81 | **12.50** |

grouped, these can only take 5.40% of the pipeline's time and this happens without merging the sub-functions. It is therefore expected that the classifier will be the one that can benefit the most from hardware acceleration.

The remaining stages have such a low contribution to the pipeline that accelerating them might not be worth while. This is especially true if the stages require a lot of data transfers, making them communication intensive. As such, the best stages for hardware acceleration are the computation intensive ones, where several operations are done on little amounts of data (meaning less data needs to be transferred from the CPU).

## 5.3.2 High-Level Synthesis

Table 5.3 shows the main HLS performance estimates obtained by Vivado HLS [15]. Appendix B.1 includes a table with additional results. These estimates use different versions of each function, applying different optimizations and code restructuring as well. The versions that are automatically generated by the compiler are identified by the optimization level used to generate them (*O1* or *O2*). The type used for these experiments was float and the target clock period selected was 10 ns. Note that some functions use the *math.h* function *pow* (or *powf* when floats are used). This function can have a significant impact on performance and resource usage, so versions without *pow* were also developed (note that the exponent is always 2, so the function can just be replaced by a mere multiplication). The *standardDeviation* stage is not included in this table, because it merely computes a square root. Since it does not use any loops, no optimizations were applied (although the estimates are still available in Appendix B.1).

Most of the versions shown here use loop pipelining or loop unrolling. For all the versions using loop pipelining, the target initiation interval (II) set was 1. Most loops were able to achieve this target, but there were a few exceptions. For details about the initiation intervals achieved, consult Appendix B.1.

Loop unrolling usually leads to better results, as is the case in both the *variance* and *signal-Magnitude* stages. The *mean* stage achieves better performance with loop pipelining, obtaining a 1.31 speedup over the baseline. The *variance* and *signalMagnitude* stages also explore code restructuring, with a version that merges the loops. Note that despite taking more clock cycles to complete than the unrolling versions, the clock period is lower. This leads to better performance in the *variance* case (7.56 speedup). For the *signalMagnitude* one, however, unrolling all loops is still better, leading to a 71.42 speedup. The table also shows the impact of the *pow* function. The *variance* stage with loop pipelining increases the speedup obtained from 3.93 to 5.10 when the *pow* (in this case, *powf*, since floats are used) function is removed.

The *classifier* stage takes a lot more clock cycles to complete than any other function. For this classifier in particular (k-NN), versions were once again developed pipelining and unrolling loops. This stage, however, is rather different, because there is a loop nest. As mentioned in Section 4.4.3, loop nests require more care during optimization, as using Vivado HLS pipelining on an outer loop will unroll the inner loops. For this classifier in particular, the inner loops are not that long, so pipelining the outer loop is feasible. Performing this optimization, along with loop unrolling on all the other loops led to the best version for k-NN. The speedup compared to the baseline is 71.42. The remaining *classifier* versions left the outer loop in the loop nest unchanged, applying optimizations to the inner loops instead. This leads to worse results, although still much better than the baseline.

For completeness once again, the feature extraction phase is grouped in a single stage to understand the performance gains that can be achieved. The versions developed here combine the best versions of each of the feature extraction stages together. Additionally, another version is considered by merging the functions when possible. This is a much trickier optimization to be done automatically. Nonetheless, it provides a significant speedup (12.50) when compared to the baseline. As such, it would be very interesting for the compiler to be able to do this sort of analysis and optimize stages using this approach.

The results shown here also include versions where other directives were used (such as the dataflow directive [104]). These, however, lead to worse results. The *signalMagnitude* stage contains a version using the dataflow directive, exlining all the loops into their own sub-functions. This version does not perform as well as the one with all loops unrolled, but the speedup obtained is still better than all the other versions.

Some versions also explore the use of loop pipelining and unrolling together, but the results are also worse than the ones obtained for the other versions. The unrolling factor in this case was 2, leading to a loop with half the number of iterations, but also pipelined. While the number of clock cycles obtained was decent, the execution time was severely impacted by the clock period, which was significantly higher than usual.

Table 5.4: High level synthesis resource usage estimates for each stage with floats. Estimates exceeding the FPGA capacity are shown in bold

| Stage | Notes | BRAM (%) | DSP (%) | FF (%) | LUT (%) |
|---|---|---|---|---|---|
| Mean | Without directives | 0.00 | 0.91 | 1.00 | 2.90 |
| | Pipeline loops (O1) | 0.00 | 0.91 | 1.00 | 2.93 |
| | Unroll loops (O2) | 0.00 | 0.91 | 1.07 | 3.73 |
| Variance | Without directives | 4.64 | 7.27 | 4.05 | 8.48 |
| | Pipeline loops | 4.64 | 7.27 | 4.29 | 8.64 |
| | Pipeline loops. Replace pow (O1) | 0.00 | 2.27 | 1.59 | 4.19 |
| | Pipeline loops. Unroll loops (f = 2). Replace pow | 1.43 | 4.55 | 2.02 | 5.76 |
| | Unroll loops. Replace pow (O2) | 0.00 | 2.27 | 1.27 | 4.51 |
| | Merge loops. Pipeline resulting loop. Replace pow | 0.00 | 2.27 | 1.26 | 3.67 |
| Signal Magnitude | Without directives | 4.64 | 7.27 | 4.15 | 8.61 |
| | Pipeline loops. Replace pow (O1) | 0.00 | 2.27 | 2.13 | 4.49 |
| | Unroll loops. Replace pow (O2) | 0.00 | 11.82 | 3.82 | 10.75 |
| | Dataflow function. Unroll loops. Replace pow | 0.00 | **106.36** | 25.87 | 100.00 |
| | Merge loops. Pipeline resulting loop. Replace pow | 0.00 | 5.91 | 1.51 | 4.70 |
| Classifier | Without directives | 4.64 | 7.73 | 4.22 | 11.23 |
| | Pipeline loops. Replace pow | 0.00 | 2.73 | 2.03 | 8.14 |
| | Unroll loops. Replace pow | 0.71 | 4.55 | 3.31 | 18.97 |
| | Pipeline nested loop. Unroll other loops. Replace pow (O2) | 0.71 | 7.73 | 5.25 | 21.58 |
| | O1 | 0.00 | 3.64 | 1.98 | 10.05 |
| Feature Extraction Grouped | Without directives | 13.93 | 23.64 | 13.04 | 32.33 |
| | Best combination of functions. Replace pow | 0.71 | 18.18 | 9.04 | 26.20 |
| | Merging functions. Replace pow | 0.71 | 15.00 | 7.14 | 22.79 |

The resource usage results for the versions shown in Table 5.3 are shown in Table 5.4. Appendix B.1 contains additional results. As expected, the unrolling versions require more resources than the ones using loop pipelining. Notice also how removing the *pow* function impacts the amount of resources needed for the *variance* stage. Another detail worth noting is how code restructuring affects resource usage. Code restructuring can reduce the amount of resources used, because loops are usually merged and this can lead to a more cost effective solution. This happened, for example, in one of the versions with feature extraction grouped.

Another key note is the fact that all the versions using the *O1* optimization level required less resources than the ones using *O2*. This is expected, as *O1* favours speed and space, while *O2* favours only speed. Therefore, despite achieving lower speedups, these versions are atractive if space is a concern.

The dataflow directive used in the *signalMagnitude* stage severely impacted the resource usage, with the estimates exceeding the FPGA DSP capacity. The results are even worse when the *pow* function is used in that case (see Appendix B.1).

The type being used also impacts both performance and resource usage. Tables 5.5 and 5.6 show the performance and resource usage results when doubles are used instead of floats. The performance and resource usage pattern is similar to the one found with floats. However, operations with doubles naturally take up more resources and require more cycles to complete. Note that in this case, the version grouping feature extraction without optimizations does not even fit in the FPGA (according to the estimates). The versions using the dataflow directive together with loop unrolling suffer from the same problem, especially when the *pow* function is used (see Appendix B.1).

The *mean* stage achieves a 1.28 speedup when the loops are pipelined. This value is close

Table 5.5: High level synthesis performance estimates for each stage with doubles. Best speedups are shown in bold

| Stage | Notes | Clock Cycles | Clock Period (ns) | Execution Time (ms) | Speedup |
|---|---|---|---|---|---|
| Mean | Without directives | 157 | 8.62 | 1.35 | - |
| | Pipeline loops (O1) | 123 | 8.62 | 1.06 | **1.28** |
| | Unroll loops (O2) | 122 | 10.36 | 1.26 | 1.07 |
| Variance | Without directives | 1545 | 9.51 | 14.70 | - |
| | Pipeline loops | 236 | 9.51 | 2.25 | 6.55 |
| | Pipeline loops. Replace pow (O1) | 176 | 8.62 | 1.52 | 9.69 |
| | Pipeline loops. Unroll loops (f = 2). Replace pow | 141 | 16.46 | 2.32 | 6.33 |
| | Unroll loops. Replace pow (O2) | 133 | 10.47 | 1.39 | 10.56 |
| | Merge loops. Pipeline resulting loop. Replace pow | 134 | 8.62 | 1.16 | **12.72** |
| Signal Magnitude | Without directives | 4669 | 9.51 | 44.42 | - |
| | Pipeline loops. Replace pow (O1) | 205 | 8.62 | 1.17 | 25.13 |
| | Unroll loops. Replace pow (O2) | 57 | 9.40 | 0.54 | **82.88** |
| | Dataflow function. Unroll loops. Replace pow | 66 | 8.62 | 0.57 | 78.05 |
| | Merge loops. Pipeline resulting loop. Replace pow | 68 | 8.62 | 0.59 | 75.76 |
| Classifier | Without directives | 555736 | 9.63 | 5353.96 | - |
| | Pipeline loops. Replace pow | 39226 | 17.95 | 703.91 | 7.61 |
| | Unroll loops. Replace pow | 39750 | 10.36 | 411.61 | 13.01 |
| | Pipeline nested loop. Unroll other loops. Replace pow (O2) | 3318 | 9.63 | 31.97 | **167.49** |
| | O1 | 38689 | 17.95 | 694.27 | 7.71 |
| Feature Extraction Grouped | Without directives | 9366 | 9.51 | 89.11 | - |
| | Best combination of functions. Replace pow | 748 | 9.40 | 7.03 | 12.67 |
| | Merging functions. Replace pow | 381 | 9.77 | 3.72 | **23.95** |

Table 5.6: High level synthesis resource usage estimates for each stage with doubles. Estimates exceeding the FPGA capacity are shown in bold

| Stage | Notes | BRAM (%) | DSP (%) | FF (%) | LUT (%) |
|---|---|---|---|---|---|
| Mean | Without directives | 0.00 | 1.36 | 3.60 | 9.44 |
| | Pipeline loops (O1) | 0.00 | 1.36 | 3.60 | 9.47 |
| | Unroll loops (O2) | 0.00 | 1.36 | 3.67 | 10.27 |
| Variance | Without directives | 25.36 | 33.18 | 18.10 | 25.80 |
| | Pipeline loops | 25.36 | 33.18 | 18.34 | 25.74 |
| | Pipeline loops. Replace pow (O1) | 0.00 | 6.36 | 4.42 | 11.19 |
| | Pipeline loops. Unroll loops (f = 2). Replace pow | 2.86 | 12.73 | 5.57 | 14.70 |
| | Unroll loops. Replace pow (O2) | 0.00 | 6.36 | 4.16 | 11.63 |
| | Merge loops. Pipeline resulting loop. Replace pow | 0.00 | 6.36 | 4.08 | 10.70 |
| Signal Magnitude | Without directives | 28.21 | 33.18 | 17.30 | 24.68 |
| | Pipeline loops. Replace pow (O1) | 4.29 | 6.36 | 3.98 | 9.24 |
| | Unroll loops. Replace pow (O2) | 0.00 | 35.45 | 12.01 | 26.18 |
| | Dataflow function. Unroll loops. Replace pow | 0.00 | **319.09** | 71.50 | **231.57** |
| | Merge loops. Pipeline resulting loop. Replace pow | 0.00 | 17.73 | 4.17 | 12.06 |
| Classifier | Without directives | 23.93 | 33.64 | 16.53 | 25.19 |
| | Pipeline loops. Replace pow | 0.00 | 6.82 | 3.25 | 12.31 |
| | Unroll loops. Replace pow | 0.71 | 9.09 | 5.37 | 28.70 |
| | Pipeline nested loop. Unroll other loops. Replace pow (O2) | 7.86 | 16.82 | 8.15 | 34.05 |
| | O1 | 0.00 | 7.73 | 3.17 | 15.97 |
| Feature Extraction Grouped | Without directives | 76.79 | **102.27** | 64.78 | **103.82** |
| | Best combination of functions. Replace pow | 1.43 | 50.91 | 29.88 | 71.87 |
| | Merging functions. Replace pow | 1.43 | 43.18 | 24.52 | 55.86 |

to the speedup obtained when using floats. The *variance* stage's best version is again the one merging the loops and pipelining the resulting loop. This version achieves a 12.72 speedup, which is considerably higher than the one obtained for floats (7.56). The *signalMagnitude* stage obtains a 82.88 speedup when all loops are unrolled. This is the same optimization applied for the version using floats, but the speedup is significantly higher. The best version grouping feature extraction achieves a 23.95 speedup, which is almost double what the best version using floats achieved.

Perhaps one of the most interesting notes to take from these results is the speedup obtained by the best *classifier* version. The best version of the *classifier* stage actually does not require many clock cycles, when compared to the best version using floats. This led to an impressive 167.49 speedup when compared to the baseline. This speedup is highly affected by the baseline itself. The baseline for the stage using doubles requires many more clock cycles than the one using floats. As such, the gap between the worst and best versions is higher when doubles are used, leading to this increase in speedup.

The clock period is overall higher when doubles are used, with some versions even failing to meet the timing constraints. Recall that the designs targeted a 10 ns clock period. Even the versions that are below the target require a significantly higher clock period than their float counterparts. Nonetheless, the overall performance and resource usage patterns remain the same, with the best versions using floats performing best when doubles are used as well.

One detail that might go unnoticed on a first look is the scale of the speedups obtained. The speedups were on average much higher when doubles were used, making the optimizations more effective.

Finally, perhaps the most important piece of information these tables provide is that optimizing the stages is not as trivial as it might seem. Notice how different stages achieved the best results using different optimization approaches. The *mean* stage achieved the best results with loop pipelining, while the *signalMagnitude* behaved better with loop unrolling. The *variance* stage achieved the best performance with the loops merged and the *classifier* stage required a mix of both pipelining and unrolling to obtain the best speedups. This clearly suggests that implementing a pragma generation strategy is not a simple task, requiring a sophisticated analysis into how optimizations should be applied. After all, the compiler has no insight into what the absolute best strategy is for each stage, so it must use one that obtains a decent result on most cases (even if not optimal). This is, according to the results, a reasonable approach, since the versions generated by the compiler (the ones annotated with *O1* and *O2*) achieved decent speedups, even if not optimal for their respective stage.

All of the versions shown in this section were compared with the FPGA baseline versions. Table 5.7 shows the speedups obtained for each of the best versions and the versions generated by the compiler when compared to the CPU implementations. All versions are able to achieve a speedup, with the *signalMagnitude* and *classifier* stages achieving the best ones. Note that most of the versions using floats obtain better speedups than the equivalent versions using doubles. The most obvious exception is present in the *classifier* stage because, as mentioned earlier, the best FPGA versions for this stage using floats were very similar to the ones using doubles. As such,

Table 5.7: Comparison between the best HLS versions and the CPU implementations, for each stage. The best speedups are shown in bold

| Stage | Type | Notes | Clock Cycles | Speedup |
|---|---|---|---|---|
| Mean | float | CPU | 139 | - |
| | | FPGA - Pipeline loops (O1) | 108 | 1.29 |
| | | FPGA - Unroll loops (O2) | 107 | **1.30** |
| | double | CPU | 154 | - |
| | | FPGA - Pipeline loops (O1) | 123 | 1.25 |
| | | FPGA - Unroll loops (O2) | 122 | **1.26** |
| Variance | float | CPU | 422 | - |
| | | FPGA - Pipeline loops. Replace pow (O1) | 139 | 3.04 |
| | | FPGA - Unroll loops. Replace pow (O2) | 116 | **3.64** |
| | | FPGA - Merge loops. Pipeline resulting loop. Replace pow | 117 | 3.61 |
| | double | CPU | 451 | - |
| | | FPGA - Pipeline loops. Replace pow (O1) | 201 | 2.24 |
| | | FPGA - Unroll loops. Replace pow (O2) | 133 | **3.39** |
| | | FPGA - Merge loops. Pipeline resulting loop. Replace pow | 134 | 3.37 |
| Standard Deviation | float | CPU | 56 | - |
| | | FPGA - No directives (O1 and O2) | 11 | **5.09** |
| | double | CPU | 65 | - |
| | | FPGA - No directives (O1 and O2) | 30 | **2.17** |
| Signal Magnitude | float | CPU | 1110 | - |
| | | FPGA - Pipeline loops. Replace pow (O1) | 174 | 6.38 |
| | | FPGA - Unroll loops. Replace pow (O2) | 35 | **31.71** |
| | double | CPU | 1449 | - |
| | | FPGA - Pipeline loops. Replace pow (O1) | 205 | 7.07 |
| | | FPGA - Unroll loops. Replace pow (O2) | 57 | **25.42** |
| Classifier | float | CPU | 51337 | - |
| | | FPGA - Pipeline nested loop. Unroll other loops. Replace pow (O2) | 3298 | **15.57** |
| | | FPGA - O1 | 37101 | 1.38 |
| | double | CPU | 81054 | - |
| | | FPGA - Pipeline nested loop. Unroll other loops. Replace pow (O2) | 3318 | **24.43** |
| | | FPGA - O1 | 38689 | 2.10 |
| Feature Extraction | float | CPU - Best combination of functions. Replace pow | 3008 | - |
| | | FPGA - Best combination of functions. Replace pow | 626 | **4.81** |
| | | CPU - Merging functions. Replace pow | 1580 | - |
| | | FPGA - Merging functions. Replace pow | 284 | **5.56** |
| | double | CPU - Best combination of functions. Replace pow | 3593 | - |
| | | FPGA - Best combination of functions. Replace pow | 748 | **4.80** |
| | | CPU - Merging functions. Replace pow | 2171 | - |
| | | FPGA - Merging functions. Replace pow | 381 | **5.70** |

since the CPU implementation using doubles requires many more clock cycles to complete than the equivalent float version, the gap is significantly higher when doubles are used.

## 5.4 Hybrid Execution

The high-level synthesis estimates reveal the performance gains each stage can have. However, actually targeting the stage to an FPGA requires more than accelerator code optimization. Communication is a key element in the equation. As such, it also requires optimizations on its own. Given the contributions of each stage to the overall pipeline, it is important to understand what the best theoretical speedup would be for each stage if they were to be targeted to an FPGA. Table 5.8 shows the estimated speedups obtained for each stage, when the stage's time is assumed to be zero. Additionally, the kernel and communication time estimates are provided for the *classifier* stage, the *signalMagnitude* stage and the stage grouping feature extraction. These estimates are

Table 5.8: Estimated and real speedups. The theoretical speedups take into account kernel (K) and communication (C) estimated clock cycles. Positive speedups are shown in bold

| **Accelerated Function** | **Notes** | **Type** | **Estimates (clock cycles)** | | **Speedup** | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | **Kernel** | **Communication** | **K = 0 C = 0** | **K = 0 C = Est** | **K = Est C = 0** | **K = Est C = Est** | **Real** |
| Mean | - | float | - | - | **1.01** | - | - | - | - |
| Variance | - | float | - | - | **1.03** | - | - | - | - |
| Standard Deviation | - | float | - | - | **1.00** | - | - | - | - |
| Signal Magnitude | - | float | 18515 | 6011556 | **1.02** | 0.85 | **1.02** | 0.85 | - |
| Classifier | - | float | 1744642 | 28040174 | **11.60** | 0.97 | **1.34** | 0.59 | **1.05** |
| | | double | 20466481 | 50432215 | **15.26** | 0.86 | **1.95** | 0.62 | **1.58** |
| Feature Extraction | Grouped | float | 331154 | 6012085 | **1.06** | 0.87 | **1.04** | 0.86 | 0.88 |
| | | double | 395692 | 6003621 | **1.04** | 0.92 | **1.03** | 0.91 | 0.91 |
| | Grouped. Merging sub-functions | float | 150236 | 6012085 | **1.03** | 0.85 | **1.02** | 0.84 | 0.89 |
| | | double | 201549 | 6003621 | **1.03** | 0.90 | **1.02** | 0.90 | 0.92 |

used to compute three more speedups. The first one uses the communication time estimate and assumes the kernel code time to be zero. The second one does the opposite while the third uses both estimates. Note that the estimates given are a worst case scenario. Finally, the real times obtained when targeting the classifier and feature extraction versions to an FPGA are given.

As expected, the *classifier* stage is the best choice for hardware acceleration. It is the only one capable of achieving any speedup at all. Notice how the versions using doubles can reach even higher speedups than the ones using floats. This is especially true for the *classifier* stage. Recall from Section 5.3.2 that the HLS estimates obtained for the best versions of this stage were very close to each other, leading to a much higher speedup for the one using doubles. The CPU version using doubles takes a lot longer to complete, which is why a bigger speedup is achieved when this data type is used.

It should also be noted that the *classifier* stage has one big flaw: communication. The stage requires a training set to be sent as a function parameter, which means it is transferred from the CPU to the FPGA on every invocation. Note that this transfer is motivated by the fact that the training set can be initialized once in the beginning, possibly using a file. As such, it needs to be used by both the CPU and the FPGA. The data transfers associated with moving the training set to the FPGA impact the performance of the accelerator, because the number of training samples is usually high. Were it not for this amount of communication, the stage had the potential to achieve an even higher speedup. Future work could also focus on improving this aspect, e.g., moving the training set closer to the computation units (i.e., the FPGA).

Note that this sort of analysis is needed in the context of hardware acceleration, because thought should be put into choosing what to accelerate. After all, some of the stages shown here contribute so little to the entire pipeline that accelerating them might not be worthwhile. Their theoretical speedups are very close to 1.00, which makes them bad choices for hardware acceleration. The classifier, however, provided a much more promising theoretical speedup, even when the high-level synthesis estimates were taken into account. It was therefore expected that this stage would be the only one to achieve a speedup in a real execution.

In the end, communication is a key factor in this equation and no matter how optimized it might be, the overall performance can still take a huge hit if too many data transfers are required between CPU and FPGA. If a stage is too communication intensive then chances are it will not be good for hardware acceleration, no matter how much optimizing efforts are focused in this section of the application. Analyzing the communication and computation intensity is vital towards understanding what to accelerate. The communication patterns are also an interesting detail to investigate, as these might help in deciding whether or not a stage is worth accelerating.

## 5.5 Summary

The previous sections showed results for several experiments in order to assert the quality of the DSL in terms of performance and productivity. It is clear that regarding productivity, the DSL does help in development. Small changes to a pipeline can be made in a couple of minutes using the DSL. Using a general purpose language, this work might take much longer. The fact that loops are left out of the DSL is also a benefit, as the code becomes a lot easier to read.

The performance of the DSL is what deserves more attention at this moment, however. As mentioned, accelerating FPGA-based applications requires optimizing the accelerator code and the CPU-FPGA data transfers (communication). The accelerator code can be improved using pragmas, which is already done by the compiler. However, as depicted earlier, code restructuring could also be used together with pragmas to obtain better designs with even less resources. As such, future work could focus on this aspect. It should, however, be made clear that the most detrimental factor in performance when targeting the CPU-FPGA system provided by SDSoC [21] at the moment is communication. The communication aspect is not taken care of at all by the compiler so any future work should focus on this.

Communication is tricky, because it sometimes requires some code restructuring on its own, as some of the communication directives supported by SDSoC [21] have certain requirements. One of the most important optimizations when it comes to communication is the streaming access pattern [105]. This pattern can be used on arrays to effectively allow the accelerator to begin execution as soon as it obtains the first element in the array marked with the pattern. This optimization, however, has certain requirements (the values must only be accessed once, for example). As such, some code transformations might be needed to apply this optimization.

This kind of analysis is difficult to execute even by hand, so implementing it in the compiler requires sophisticated analysis. Even the code transformations performed to improve the accelerator code are sometimes complex, so implementing them might not be straightforward. Despite this, the fact that the compiler knows about the domain being worked upon could facilitate this process. Some assumptions based on the data analytics domain can be taken, allowing code transformations to be performed more easily.

The communication section can also be improved by moving certain variables closer to the computation units (i.e., the FPGA). A candidate for this type of optimization is the training set in the *classifier* stage, as mentioned earlier.

The grouping of the feature extraction stages in a single stage showcases another important detail that could be focused in the future. Many stages contribute too little to the overall pipeline, so merging them and accelerating a group of stages is usually better. This is an interesting feature that could be supported by developing a new pragma in the compiler.

The compiler could even go as far as having a mode where it decides on its own what is worth accelerating. This sort of analysis could take both accelerator code and communication overhead into account, making sure only the best stages are accelerated. This would also allow stages to be grouped automatically, without burdening the developer with such tasks.

One other detail that could be explored is constraint relaxation. Many classifiers in machine learning can improve performance significantly if certain constraints are relaxed. For example, the size of the window during feature extraction in the example used throughout this chapter was 18. An interesting avenue to pursue would be to see how changing this value (to 16, for instance) would impact the overall performance. Note that this sort of code transformation may decrease the classifier accuracy. However, this decrease is often marginal, making the optimization appealing due to the performance increase.

Although the FPGA is the focus of this research, the CPU versions themselves could also be improved. This would allow for a better comparison to be made since more effort was put in optimizing the FPGA in these experiments. A better approach would be to use a manually optimized CPU version, instead of the one generated by the compiler. Moreover, the compiler itself could be enhanced with CPU optimizations, using OpenMP, for example.

Overall, the results have shown that for this domain, hardware acceleration can be effective. However, much work needs to be done in optimization. The introduction of a new DSL is good on that regard, because the compiler can take care of the optimizations, leaving the developer out of it. Moreover, even if the developer is not entirelly happy with the generated code, it can act as a skeleton on which more optimizations can be introduced. As such, even if the DSL does not always generate the most efficient solutions, it helps in creating a baseline to optimize on. This can be very helpful when a developer is still prototyping a pipeline, as the most simple changes to the program might require hours of work using a general purpose language.

# Chapter 6

# Conclusions

This chapter concludes this dissertation. A brief review on the goals achieved is given, providing answers to the proposed research questions as well. The chapter ends with a discussion on future work.

## 6.1 Goals

The main goal of this dissertation was to develop a new DSL for data analytics, capable of targeting FPGA-based systems. A case study was used to guide the development of the DSL, allowing it to be oriented towards the data analytics domain. The DSL developed contains several domain-specific constructs that improve the development process, especially in the prototyping phase, as small changes in a given program require little effort, as opposed to what usually happens using a general purpose language.

The compiler developed is also able to target FPGAs via HLS, which was the main goal after all. The compiler is additionally capable of optimizing code to run on the hardware accelerator, allowing data analytics systems to benefit from the performance levels associated with FPGA execution.

## 6.2 Research Questions

Four research questions were identified in Chapter 1. This section provides answers to those questions.

A.1 *Can data analytics applications benefit from standalone FPGA execution using HLS?*

Taking into account the results presented in Chapter 5, data analytics applications (at least in the context of human activity recognition) can certainly benefit from standalone FPGA execution using HLS. The amount of parallel work is significant, allowing these devices to exploit it to improve performance. The case study used for the evaluation was a human activity recognition

(HAR) system and the HLS performance experiments show that significant speedups can be obtained, even when compared to the CPU implementations. As such, in the context of HAR, the use of a standalone FPGA seems beneficial. To better understand how data analytics systems behave in general in this scenario, additional experiments would need to be made, not just in the context of HAR, but in other contexts as well.

A.2 *Can data analytics applications benefit from mixed CPU-FPGA execution using HLS?*

As mentioned in Chapter 5, mixed CPU-FPGA execution is highly affected by the amount of communication performed. In a standalone FPGA scenario, performance is affected essentially by the hardware accelerator itself. In a mixed CPU-FPGA, communication is an additional factor that requires optimization on its own, in order to improve performance. The results in Chapter 5 show that despite the communication factor, CPU-FPGA solutions using HLS can still outperform the CPU (in the context of HAR), although this does not happen nearly as often as it did for standalone FPGA solutions. Communication is the key element here, as it can be detrimental to the overall performance in the pipeline, even when optimized. If a stage has too many data transfers, this significantly hurts performance, making the stage less appealing for hardware acceleration. Nonetheless, the stage with the highest contribution in the case study used for evaluation was able to achieve speedups when it was hardware accelerated. As such, HAR systems seem like a good fit for mixed CPU-FPGA execution. Again, to understand the behavior of data analytics systems in general, additional experiments using different case studies would need to be made.

It should also be mentioned that for the evaluated implementations, the CPU executes at 667 MHz, while the hardware accelerator executes at 100 MHz. Therefore, better improvements might be achieved with the hardware accelerator using higher clock frequencies.

B.1 *Can data analytics applications be targeted to FPGAs using a DSL approach?*

As shown throughout this dissertation, a DSL can effectively be used to program data analytics systems on a mixed CPU-FPGA system. The DSL developed is capable of accelerating stages to an FPGA, providing valid C code that can be input to Vivado HLS. Moreover, the DSL leaves any hardware specific details away from the user, allowing quick algorithm prototyping. This is certainly a plus as it allows software developers with little to no knowledge on hardware design to target their data analytics applications to an FPGA-based system.

B.2 *Can data analytics applications improve performance using a new DSL for FPGAs?*

Based on the results obtained in Chapter 5, data analytics applications (at least in the context of HAR) can certainly improve performance when targeted to an FPGA. The question is whether or not this performance gain can be obtained without burdening the developer with hardware specific details. That is, to answer this question, one needs to understand whether or not a compiler would be able to generate good enough versions for the performance to be better on an FPGA.

The results shown throughout Chapter 5 clearly indicate that designing a pragma generation strategy capable of generating the best stages every time is most likely impossible, because different stages require different optimizations most of the time. However, as long as the compiler is capable of finding a decent version for every stage, the results will most likely be good enough.

The current strategy implemented in the compiler is not very robust. However, the HLS experiments conducted on the HAR case study showed that even though achieving the best versions for each stage seems too difficult, most of the other versions can still acquire significant speedups when compared to the baselines. As such, it is expected that using a DSL for FPGAs is a feasible solution to improve the performance of data analytics applications.

## 6.3 Future Work

Although the developed DSL is capable of targeting FPGAs, much work can still be done on both the compiler and the DSL itself. The pragma generation strategy currently supported is rather simple, leading to less than optimal results. Despite the performance improvement already achieved, much more can be done, especially regarding communication. As mentioned, the DSL compiler currently focuses only on the accelerator code, generating HLS pragmas to improve performance. However, the data transfers can sometimes be detrimental to the overall performance. As such, future work on the compiler could focus on generating communication directives to optimize data transfers. Moreover, the accelerator code itself can be further optimized if a more robust pragma generation policy is developed. Code restructuring techniques can be used for this purpose, allowing the accelerator code to not only achieve better performance, but perhaps fewer resources used.

The DSL itself can also be extended with new features. As mentioned in Chapter 4, the DSL currently supports only one classifier: k-NN. New classifiers could be introduced and these would bring new challenges, as many algorithms throughout machine learning have very different characteristics. The ability to train models for algorithms that require this step could also be an interesting enhancement to the DSL. Finally, the DSL could turn its focus to more realistic scenarios. The current version is focused on prototyping solutions, allowing data to be read from files. In a real embedded system, however, data might come from sensors. Support for this type of real world situations would also be an interesting avenue to pursue, allowing data analytics systems to be entirely written using the DSL.

# References

[1] John von Neumann. First draft of a report on the EDVAC. Report, Moore School of Electrical Engineering, University of Pennsylvania, 1945.

[2] J. von Neumann. First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[3] S. Patel and W. W. Hwu. Accelerator Architectures. *IEEE Micro*, 28(4):4–12, 2008.

[4] J. W. Choe, A. Nikoozadeh, Oralkan Ö, and B. T. Khuri-Yakub. GPU-Based Real-Time Volumetric Ultrasound Image Reconstruction for a Ring Array. *IEEE Transactions on Medical Imaging*, 32(7):1258–1264, 2013.

[5] C. Chung, C. Liu, and D. Lee. FPGA-based accelerator platform for big data matrix processing. In *2015 IEEE International Conference on Electron Devices and Solid-State Circuits (EDSSC)*, pages 221–224. IEEE, 2015.

[6] K. Neshatpour, M. Malik, M. A. Ghodrat, and H. Homayoun. Accelerating Big Data Analytics Using FPGAs. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 164–164, Washington, DC, USA, 2015. IEEE Computer Society.

[7] H. Chen, Y. Chen, and D. H. Summerville. A Survey on the Application of FPGAs for Network Infrastructure Security. *IEEE Communications Surveys & Tutorials*, 13(4):541–561, 2011.

[8] A. Putnam et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, pages 13–24, Piscataway, NJ, USA, 2014. IEEE Press.

[9] Andrew Putnam et al. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News - ISCA '14*, 42(3):13–24, 2014.

[10] A. Moore and R. Wilson. *FPGAs For Dummies*. John Wiley & Sons, Inc., 2nd intel special edition, 2017.

[11] João Moreira, Andre de Carvalho, and Tomas Horvath. *A General Introduction to Data Analytics*. Wiley-Interscience, 1st edition, 2018.

[12] Ethem Alpaydin. *Introduction to Machine Learning*. The MIT Press, 3rd edition, 2014.

[13] Alok N. Choudhary, Daniel Honbo, Prabhat Kumar, Berkin Özisikyilmaz, Sanchit Misra, and Gokhan Memik. Accelerating data mining workloads: current approaches and future challenges in system architecture design. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.*, 1(1):41–54, 2011.

[14] Sumit K. Ghosh. *Hardware Description Languages: Concepts and Principles*. Wiley-IEEE Press, 1st edition, 1999.

[15] Xilinx Inc. Vivado High-Level Synthesis. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html, 2019. Last accessed February 1st, 2019.

[16] S. Windh, Ma Xiaoyin, R. J. Halstead, P. Budhkar, Z. Luna, O. Hussaini, and W. A. Najjar. High-level language tools for reconfigurable computing. *Proceedings of the IEEE*, 103(3):390–408, 2015.

[17] N. Kapre and S. Bayliss. Survey of domain-specific languages for FPGA computing. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–12, Lausanne, Switzerland, 2016. IEEE.

[18] I. Portugal, P. Alencar, and D. Cowan. A Preliminary Survey on Domain-Specific Languages for Machine Learning in Big Data. In *2016 IEEE International Conference on Software Science, Technology and Engineering (SWSTE)*, pages 108–110. IEEE, 2016.

[19] I. Portugal, P. S. C. Alencar, and D. D. Cowan. A Survey on Domain-Specific Languages for Machine Learning in Big Data. Report, University of Waterloo, Canada, 2016. arXiv preprint arXiv:1602.07637.

[20] D. F. Bacon, R. Rabbah, and S. Shukla. FPGA programming for the masses. *ACM Queue*, 11(2):40, 2013.

[21] Xilinx Inc. SDSoC Development Environment. https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html, 2019. Last accessed June 19th, 2019.

[22] 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, 2015.

[23] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[24] S. M. S. Trimberger. Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology: This Paper Reflects on How Moore's Law Has Driven the Design of FPGAs Through Three Epochs: the Age of Invention, the Age of Expansion, and the Age of Accumulation. *IEEE Solid-State Circuits Magazine*, 10(2):16–29, 2018.

[25] João M. P. Cardoso and Markus Weinhardt. High-Level Synthesis. In *FPGAs for Software Programmers*, pages 23–47. Springer International Publishing, 2016.

[26] D. D. Gajski and L. Ramachandran. Introduction to high-level synthesis. *IEEE Design & Test of Computers*, 11(4):44–54, 1994.

[27] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. An Introduction to High-Level Synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, 2009.

[28] 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, 2016.

[29] Maxeler Technologies. MaxCompiler | Maxeler Technologies. https://www.maxeler.com/products/software/maxcompiler/, n.d. Last accessed February 1st, 2019.

[30] Maxeler Technologies. MaxCompiler White Paper. Technical report, Maxeler Technologies, 2011.

[31] Maxeler Technologies. Programming MPC Systems White Paper. Technical report, Maxeler Technologies, 2013.

[32] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[33] Imprint. Hipacc: Overview. https://hipacc-lang.org/, 2018. Last accessed February 1st, 2019.

[34] R. Membarth, O. Reiche, F. Hannig, and J. Teich. Code generation for embedded heterogeneous architectures on android. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 3001 Leuven, Belgium, Belgium, 2014. European Design and Automation Association.

[35] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert. Generating Device-specific GPU Code for Local Operators in Medical Imaging. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 569–581. IEEE, 2012.

[36] O. Reiche, M. Schmid, F. Hannig, R. Membarth, and J. Teich. Code generation from a domain-specific language for C-based HLS of hardware accelerators. In *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10. IEEE, 2014.

[37] Richard Membarth, Oliver Reiche, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. HIPAcc: A Domain-Specific Language and Compiler for Image Processing. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):210–224, 2016.

[38] M. A. Özkan, O. Reiche, F. Hannig, and J. Teich. FPGA-based accelerator design from a domain-specific language. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–9. IEEE, 2016.

[39] O. Reiche, M. A. Özkan, R. Membarth, J. Teich, and F. Hannig. Generating FPGA-based image processing accelerators with Hipacc: (Invited paper). In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1026–1033. IEEE, 2017.

[40] Jonathan Ragan-Kelley. Halide. http://halide-lang.org/, n.d. Last accessed February 2nd, 2019.

[41] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics (TOG)*, 31(4):1–12, 2012.

[42] Jonathan Ragan-Kelley. *Decoupling Algorithms from the Organization of Computation for High Performance Image Processing*. Phd thesis, Massachusetts Institute of Technology, USA, 2014.

[43] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices - PLDI '13*, 48(6):519–530, 2013.

[44] 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*, pages 519–530, New York, NY, USA, 2013. ACM.

[45] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)*, 35(4):1–11, 2016.

[46] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming Heterogeneous Systems from an Image Processing DSL. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):1–25, 2017.

[47] Multicore Computing Lab, CSA, IISc. PolyMage. http://mcl.csa.iisc.ac.in/polymage.html, n.d. Last accessed February 1st, 2019.

[48] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. PolyMage: Automatic Optimization for Image Processing Pipelines. *ACM SIGARCH Computer Architecture News - ASPLOS'15*, 43(1):429–443, 2015.

[49] 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, New York, NY, USA, 2016. ACM.

[50] Abhinav Jangda and Uday Bondhugula. An effective fusion and tile size model for optimizing image processing pipelines. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 261–275, New York, NY, USA, 2018. ACM.

[51] Abhinav Jangda and Uday Bondhugula. An effective fusion and tile size model for optimizing image processing pipelines. *ACM SIGPLAN Notices - PPoPP '18*, 53(1):261–275, 2018.

[52] The OpenSPL Consortium. OpenSPL. http://www.openspl.org/, 2019. Last accessed February 2nd, 2019.

[53] The OpenSPL Consortium. OpenSPL: Revealing the Power of Spatial Computing. Technical report, The OpenSPL Consortium, 2013.

[54] T Becker, O Mencer, and G Gaydadjiev. Spatial programming with OpenSPL. In *FPGAs for Software Programmers*, pages 81–95. Springer International Publishing, 2016.

[55] Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Michael Wu, Anand R. Atreya, Kunle Olukotun, Tiark Rompf, and Martin Odersky. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, pages 609–616, USA, 2011. Omnipress.

[56] Stanford University. OptiML — Welcome. http://stanford-ppl.github.io/Delite/optiml/, 2011. Last accessed February 1st, 2019.

[57] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. *ACM SIGPLAN Notices - PPoPP '11*, 46(8):35–46, 2011.

[58] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 89–100, Washington, DC, USA, 2011. IEEE Computer Society.

[59] Stanford University. Delite — Welcome. http://stanford-ppl.github.io/Delite/index.html, 2012. Last accessed February 1st, 2019.

[60] H. Lee, K. Brown, A. Sujeeth, H. Chafi, T. Rompf, M. Odersky, and K. Olukotun. Implementing Domain-Specific Languages for Heterogeneous Parallel Computing. *IEEE Micro*, 31(5):42–53, 2011.

[61] Tiark Rompf, Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Building-Blocks for Performance Oriented DSLs. In *DSL '11: IFIP Working Conference on Domain-Specific Languages*, 2011.

[62] N. George, H. Lee, D. Novo, T. Rompf, K. J. Brown, A. K. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne. Hardware system synthesis from Domain-Specific Languages. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2014.

[63] Google, Inc. Tensorflow. https://www.tensorflow.org/, n.d. Last accessed February 1st, 2019.

[64] Martín Abadi et al. TensorFlow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association.

[65] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[66] Martín Abadi et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. https://www.tensorflow.org/, 2015. Software available from tensorflow.org.

[67] Norm Jouppi. Google supercharges machine learning tasks with TPU custom chip. https://cloud.google.com/blog/products/gcp/google-supercharges-machine-learning-tasks-with-custom-chip, 2016. Last accessed February 1st, 2019.

[68] Daniel H. Noronha, Bahar Salehpour, and Steven J. E. Wilton. LeFlow: Enabling Flexible FPGA High-Level Synthesis of Tensorflow Deep Neural Networks, 2018. arXiv preprint arXiv:1807.05317.

[69] 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, pages 33–36, New York, NY, USA, 2011. ACM.

[70] LISA lab. Welcome — Theano 1.0.0 documentation. http://deeplearning.net/software/theano/, 2017. Last accessed February 1st, 2019.

[71] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph P. Turian, David Warde-Farley, and Yoshua Bengio. Theano : A CPU and GPU Math Compiler in Python. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, 2010.

[72] Rami Al-Rfou et al. Theano: A Python framework for fast computation of mathematical expressions, 2016. arXiv preprint arXiv:1605.02688.

[73] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[74] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 65–74, New York, NY, USA, 2017. ACM.

[75] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, New York, NY, USA, 2014. ACM.

[76] R. DiCecco, G. Lacey, J. Vasiljevic, P. Chow, G. Taylor, and S. Areibi. Caffeinated FPGAs: FPGA framework For Convolutional Neural Networks. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 265–268. IEEE, 2017.

[77] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS-W*, 2017.

[78] Norman A. Rink, Immo Huismann, Adilla Susungi, Jeronimo Castrillon, Jörg Stiller, Jochen Fröhlich, and Claude Tadonki. CFDlang: High-level code generation for high-order methods in fluid dynamics. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, pages 1–10, New York, NY, USA, 2018. ACM.

[79] Adilla Susungi, Norman A. Rink, Albert Cohen, Jeronimo Castrillon, and Claude Tadonki. Meta-programming for cross-domain tensor optimizations. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 79–92, New York, NY, USA, 2018. ACM.

[80] Sven Karol, Tobias Nett, Pietro Incardona, Nesrine Khouzami, Jerónimo Castrillón, and Ivo F. Sbalzarini. A Language and Development Environment for Parallel Particle Methods. In *V. International Conference on Particle-based Methods - Fundamentals and Applications*. International Center for Numerical Methods in Engineering (CIMNE), 2017.

[81] Sven Karol, Tobias Nett, Jeronimo Castrillon, and Ivo F. Sbalzarini. A Domain-Specific Language and Editor for Parallel Particle Methods. *ACM Transactions on Mathematical Software (TOMS)*, 44(3):1–32, 2018.

[82] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom: compiling high-level image processing code into hardware pipelines. *ACM Transactions on Graphics (TOG)*, 33(4):1–11, 2014.

[83] 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 (TRETS) - Special Section on FCCM 2016 and Regular Papers*, 11(1):7:1–7:24, 2018.

[84] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, Xiong Jianxin, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[85] P. D' Alberto, P. A. Milder, A. Sandryhaila, F. Franchetti, J. C. Hoe, J. M. F. Moura, M. Puschel, and J. R. Johnson. Generating FPGA-Accelerated DFT Libraries. In *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, pages 173–184, Washington, DC, USA, 2007. IEEE Computer Society.

[86] Chia-Jui Hsu, Ming-Yung Ko, and Shuvra S. Bhattacharyya. Software Synthesis from the Dataflow Interchange Format. In *Proceedings of the 2005 Workshop on Software and Compilers for Embedded Systems*, SCOPES '05, pages 37–49, New York, NY, USA, 2005. ACM.

[87] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. Mcrae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. Firedrake: Automating the Finite Element Method by Composing Abstractions. *ACM Transactions on Mathematical Software (TOMS)*, 43(3):24:1–24:27, 2016.

[88] Martin S. Alnæs, Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. Unified Form Language: A Domain-specific Language for Weak Formulations of Partial Differential Equations. *ACM Transactions on Mathematical Software (TOMS)*, 40(2):9:1–9:37, 2014.

[89] David Garlan, Robert Monroe, and David Wile. Acme: An Architecture Description Interchange Language. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '97, page 7. IBM Press, 1997.

[90] Carlos Canal, Ernesto Pimentel, and José M. Troya. Specification and Refinement of Dynamic Software Architectures. In *Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, pages 107–126, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[91] Chen Li, Hong-Ji Yang, Mei-Yu Shi, and Wei Zhu. xBreeze/ADL: A language for software architecture specification and analysis. *International Journal of Automation and Computing*, 13(6):552–564, 2016.

[92] Jocelyn Serot, Francois Berry, and Sameer Ahmed. Implementing Stream-Processing Applications on FPGAs: A DSL-Based Approach. In *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications*, FPL '11, pages 130–137, Washington, DC, USA, 2011. IEEE Computer Society.

[93] Jocelyn Sérot, François Berry, and Sameer Ahmed. CAPH: A language for implementing stream-processing applications on FPGAs. In *Embedded Systems Design with FPGAs*, pages 201–224. Springer, New York, NY, 2013.

[94] Roger D. Chamberlain. Assessing User Preferences in Programming Language Design. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, pages 18–29, New York, NY, USA, 2017. ACM.

[95] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2006.

[96] Xilinx Inc. HLS Pragmas. https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/okr1504034364623-2.html, 2019. Last accessed June 29th, 2019.

[97] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[98] ANTLR / Terence Parr. ANTLR. https://www.antlr.org/, 2014. Last accessed February 1st, 2019.

[99] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, second edition edition, 2013.

[100] Sven Efftinge and Miro Spoenemann. Xtext - Language Engineering Made Easy! https://www.eclipse.org/Xtext/, n.d. Last accessed February 1st, 2019.

[101] Oracle. JavaCC - The Java Parser Generator. https://javacc.org/, n.d. Last accessed February 1st, 2019.

[102] Yun Fu. *Human Activity Recognition and Prediction*. Springer Publishing Company, Incorporated, 1st edition, 2015.

[103] Attila Reiss and Didier Stricker. Introducing a New Benchmarked Dataset for Activity Monitoring. In *Proceedings of the 2012 16th Annual International Symposium on Wearable Computers (ISWC)*, ISWC '12, pages 108–109, Washington, DC, USA, 2012. IEEE Computer Society.

[104] Xilinx Inc. Data Flow Pipelining. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/sdsoc_doc/topics/calling-coding-guidelines/concept_data_flow.html, 2019. Last accessed June 29th, 2019.

[105] Xilinx Inc. SDS Pragmas. https://www.xilinx.com/html_docs/xilinx2019_1/sdsoc_doc/sds-pragmas-nmc1504034362475.html#pna1504034362661, 2019. Last accessed June 29th, 2019.

# Appendix A

# DSL and Compiler details

This appendix contains additional information about the DSL, including the grammar and the types supported.

## A.1 Grammar

The DSL grammar is shown in Listing A.1.

```
<Program> -> <Declaration>+
<Declaration> -> <ConstDeclaration>
               | <AliasDeclaration>
               | <StageDeclaration>
               | <InterfaceDeclaration>
<ConstDeclaration> -> 'const' <Identifier> '=' <Expression> ';'
<AliasDeclaration> -> 'alias' <RootType> <Identifier> ';'
<StageDeclaration> -> 'stage' <Identifier> ('with' 'overlapping' <
    OverlappingExpression>)? '{' <StageProperty>+ '}'
<OverlappingExpression> -> <Identifier>
                         | <DecimalLiteral>
<StageProperty> -> <Input>
                 | <Output>
                 | <Behavior>
                 | <Setup>
<InterfaceDeclaration> -> 'interface' '{' <From> <To> '}' ';'?
<Input> -> 'input' <InputDeclaration>
         | 'input' '{' <InputDeclaration>+ '}' ';'?
<InputDeclaration> -> 'fillable'? <VariableDeclaration> ';'
<Behavior> -> 'behavior' <Statement>
            | 'behavior' '{' <Statement>+ '}' ';'?
<Output> -> 'output' <OutputDeclaration>
          | 'output' '{' <OutputDeclaration>+ '}' ';'?
<OutputDeclaration> -> <VariableDeclaration> ';'
<Setup> -> 'setup' <Statement>
         | 'setup' '{' <Statement>+ '}' ';'?
```

```
<From> -> 'from' <SimplePipelineNodeFrom> ';'
<To> -> 'to' <SimplePipelineNodeTo> ';'
      | 'to' '{' <PipelineNodeTo>+ '}' ';'?
<PipelineNodeTo> -> (<SourceIndexes> '->')? <SimplePipelineNodeTo> ';'
<SourceIndexes> -> DecimalLiteral ('..' <DecimalLiteral>)? (',' <SourceIndexes>)?
<SimplePipelineNodeFrom> -> <InterfaceFile>
                          | <InterfaceStage>
<SimplePipelineNodeTo> -> <InterfaceFile>
                        | <InterfaceStage>
                        | 'stdout'
<InterfaceFile> -> 'file' <FileReadTemplateArguments>? <StringLiteral> FileReadMode
    ?
<FileReadTemplateArguments> -> '<' <RootType> '>'
<FileReadMode> -> 'repeat'
                | 'header'
<InterfaceStage> -> 'stage'? <Identifier> ('.' <Identifier>) ('with' 'overlapping'
    <OverlappingExpression>)?
<Statement> -> <VariableDeclarationWithOptionalDefinition>
             | <Expression> ';'
<Expression> -> <AssignmentExpression>
<AssignmentExpression> -> <ConditionalExpression>
                        | <UnaryExpression> <AssignmentOperator> <
                            AssignmentExpression>
<AssignmentOperator> -> '='
                      | '*='
                      | '/='
                      | '%='
                      | '+='
                      | '-='
                      | '<<='
                      | '>>='
                      | '&='
                      | '^='
                      | '|='
<ConditionalExpression> -> <LogicalOrExpression> ('?' <Expression> ':' <
    ConditionalExpression>)?
<LogicalOrExpression> -> <LogicalAndExpression>
                       | <LogicalOrExpression> '||' <LogicalAndExpression>
<LogicalAndExpression> -> <InclusiveOrExpression>
                        | <LogicalAndExpression> '&&' <InclusiveOrExpression>
<InclusiveOrExpression> -> <ExclusiveOrExpression>
                         | <InclusiveOrExpression> '|' <ExclusiveOrExpression>
<ExclusiveOrExpression> -> <AndExpression>
                         | <ExclusiveOrExpression> '(+)' <AndExpression>
<AndExpression> -> <EqualityExpression>
                 | <AndExpression> '&' <EqualityExpression>
<EqualityExpression> -> <RelationalExpression>
                      | <EqualityExpression> '==' <RelationalExpression>
                      | <EqualityExpression> '!=' <RelationalExpression>
```

```
<RelationalExpression> -> <ShiftExpression>
                        | <RelationalExpression> '<' <ShiftExpression>
                        | <RelationalExpression> '>' <ShiftExpression>
                        | <RelationalExpression> '<=' <ShiftExpression>
                        | <RelationalExpression> '>=' <ShiftExpression>
<ShiftExpression> -> <AdditiveExpression>
                   | <ShiftExpression> '<<' <AdditiveExpression>
                   | <ShiftExpression> '>>' <AdditiveExpression>
<AdditiveExpression> -> <MultiplicativeExpression>
                      | <AdditiveExpression> '+' <MultiplicativeExpression>
                      | <AdditiveExpression> '-' <MultiplicativeExpression>
<MultiplicativeExpression> -> <CastExpression>
                            | <MultiplicativeExpression> '*' <CastExpression>
                            | <MultiplicativeExpression> '/' <CastExpression>
                            | <MultiplicativeExpression> '%' <CastExpression>
<CastExpression> -> <UnaryExpression>
                  | '(' <Type> ')' <CastExpression>
<UnaryExpression> -> <PostfixExpression>
                   | '++' <UnaryExpression>
                   | '--' <UnaryExpression>
                   | <UnaryOperator> <CastExpression>
<UnaryOperator> -> '-'
                 | '!'
                 | '~'
<PostfixExpression> -> <MainExpression>
                     | <PostfixExpression> '[' <Expression> ']'
                     | <PostfixExpression> '++'
                     | <PostfixExpression> '--'
<MainExpression> -> <Identifier>
                  | <Literal>
                  | '(' <Expression> ')'
                  | 'sum' '(' <Expression> ')'
                  | 'avg' '(' <Expression> ')'
                  | 'sqrt' '(' <Expression> ')'
                  | <MainExpression> '^' <MainExpression>
                  | 'readTrainingSet' <ReadTrainingSetTemplateArguments>? '(' <StringLiteral> ',' <Expression> ',' <Expression> ')'
                  | 'readTrainingSetToHeader' <ReadTrainingSetTemplateArguments>? '(' <StringLiteral> ',' <Expression> ',' <Expression> ')'
                  | 'knn' '(' <Expression> ',' <Expression> ',' <Expression> ',' <Expression> ')'
<ReadTrainingSetTemplateArguments> -> '<' <RootType> '>'
<VariableDeclaration> -> <Type> <Identifier>
<VariableDeclarationWithOptionalDefinition> -> <Type> <Identifier> ('=' <Expression>)? ';'
<Type> -> <RootType> ('[' <Expression> ']')*
<RootType> -> <PrimitiveType>
            | <Identifier>
<PrimitiveType> -> ('signed'? 'integer' | 'signed')
```

```
                 | (’unsigned’? ’integer’)
                 | (’long’ ’integer’? | ’signed’ ’long’ ’integer’?)
                 | (’unsigned’ ’long’ ’integer’?)
                 | (’short’ ’integer’? | ’signed’ ’short’ ’integer’?)
                 | (’unsigned’ ’short’ ’integer’?)
                 | ’float’
                 | ’double’
                 | ’char’
                 | ’signed’ ’char’
                 | ’unsigned’ ’char’
                 | ’boolean’
<Literal> -> <DecimalLiteral>
           | <FloatLiteral>
           | <BooleanLiteral>
           | <CharacterLiteral>
<BooleanLiteral> -> ’true’
                  | ’false’
<Identifier> -> <Letter> <LetterOrDigit>*
<DecimalLiteral> -> Digit+
<FloatLiteral> -> <Digit>+ ’.’ <Digit>+
<CharacterLiteral> -> ’’’ <CharacterLiteralContent> ’’’
<StringLiteral> -> ’"’ <StringLiteralContent>* ’"’
<Letter> -> [a-zA-Z_]
<Digit> -> [0-9]
<LetterOrDigit> -> <Letter>
                 | <Digit>
<CharacterLiteralContent> -> [^’\\\r\n]
                           | <EscapeSequence>
<StringLiteralContent> -> [^"\\\r\n]
                        | <EscapeSequence>
                        | ’\\\r\n’
                        | ’\\\n’
<EscapeSequence> -> ’\\’ [’"?abfnrtv\\]
```

Listing A.1: DSL Grammar

## A.2 Types

The DSL types are shown in Table A.1. These types resemble most of the C types, with the exception of the boolean type. The boolean type is mapped to a C char. The remaining types are mapped to their C counterparts.

## A.3 DSL Compiler Usage

The DSL Compiler can be invoked using the command in Listing A.2. There are also a few options that can be used. These are showcased in Table A.2.

Table A.1: DSL types

| Type | Description |
|---|---|
| **boolean** | Boolean type. 1 byte long. Maps to a C char |
| **unsigned char** | Unsigned character type. 1 byte long |
| **signed char** | Signed character type. Can be either signed or unsigned. 1 byte long |
| **char** | Character type. 1 byte long |
| **unsigned short** | Unsigned short type. 2 bytes long |
| **short** | Signed short type. 2 bytes long |
| **unsigned int** | Unsigned integer type. 4 bytes long |
| **int** | Signed integer type. 4 bytes long |
| **unsigned long** | Unsigned long type. 8 bytes long |
| **long** | Signed long type. 8 bytes long |
| **float** | Floating point type. 4 bytes long |
| **double** | Floating point type. 8 bytes long |

```
java -jar MyDSLCompiler.jar [options] <filename>
```

Listing A.2: DSL Compiler invocation

Table A.2: DSL compiler options

| Option | Long Alternative | Arguments | Description |
|---|---|---|---|
| -h | –help | N/A | Display help information. This includes a description of the available options |
| -t | –target | *chosenTarget*={*CPU*,*FPGA*} | Sets the target. The choice can be either CPU or FPGA |
| -m | –measure | N/A | Adds time measuring code to the generated pipeline |
| -O0 | N/A | N/A | Use no optimizations |
| -O1 | N/A | N/A | Optimize for speed and space |
| -O2 | N/A | N/A | Optimize for speed only |

# Appendix B

# Full Experimental Results

This appendix contains experimental results that are not fully shown in Chapter 5. As some of the tables used throughout that chapter were simplified, the full tables are shown here.

## B.1 High-Level Synthesis

Tables B.1 and B.2 show the full performance results when floats are used. This includes all the versions developed. Table B.3 displays the initiation intervals obtained for the loops in the versions that used loop pipelining. Table B.4 shows the resource usage estimates for all the versions.

Tables B.5 and B.6 show the performance results for the versions using doubles. The initiation intervals obtained for these versions are shown in Table B.7. Finally, Table B.8 displays the resource usage estimates obtained for these versions.

Table B.1: High level synthesis performance estimates for the *mean*, *variance*, *standardDeviation* and *signalMagnitude* stages with floats

| Stage | Notes | Latency (Clock cycles) | | Interval (Clock cycles) | | Clock Period (ms) | Execution Time (ms) | | Speedup | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Min | Max | Min | Max | | Min | Max | Min | Max |
| Mean | No directives | 142 | 142 | 142 | 142 | 7.26 | 1.03 | 1.03 | - | - |
| | Pipeline loops (O1) | 108 | 108 | 108 | 108 | 7.26 | 0.78 | 0.78 | 1.31 | **1.31** |
| | Pipeline loops. Unroll loops (f=2) | 107 | 107 | 107 | 107 | 9.38 | 1.00 | 1.00 | 1.03 | 1.03 |
| | Unroll loops (O2) | 107 | 107 | 107 | 107 | 9.38 | 1.00 | 1.00 | 1.03 | 1.03 |
| Variance | No directives | 684 | 684 | 684 | 684 | 9.39 | 6.42 | 6.42 | - | - |
| | Pipeline loops | 174 | 174 | 174 | 174 | 9.39 | 1.63 | 1.63 | 3.93 | 3.93 |
| | Pipeline loops. Replace pow (O1) | 157 | 157 | 157 | 157 | 8.02 | 1.26 | 1.26 | 5.10 | 5.10 |
| | Pipeline loops. Unroll loops (f=2) | 139 | 139 | 139 | 139 | 14.51 | 2.02 | 2.02 | 3.18 | 3.18 |
| | Pipeline loops. Unroll loops (f=2) Replace pow | 122 | 122 | 122 | 122 | 14.51 | 1.77 | 1.77 | 3.63 | 3.63 |
| | Unroll loops | 132 | 132 | 132 | 132 | 9.50 | 1.25 | 1.25 | 5.12 | 5.12 |
| | Unroll loops. Replace pow (O2) | 116 | 116 | 116 | 116 | 9.49 | 1.10 | 1.10 | 5.83 | 5.83 |
| | Dataflow function. Pipeline loops | 177 | 177 | 109 | 109 | 9.39 | 1.66 | 1.66 | 3.86 | 3.86 |
| | Dataflow function. Pipeline loops. Replace pow | 159 | 159 | 109 | 109 | 8.02 | 1.28 | 1.28 | 5.03 | 5.03 |
| | Dataflow function. Unroll loops | 141 | 141 | 106 | 106 | 9.39 | 1.32 | 1.32 | 4.85 | 4.85 |
| | Dataflow function. Unroll loops. Replace pow | 124 | 124 | 106 | 106 | 9.38 | 1.16 | 1.16 | 5.52 | 5.52 |
| | Merge loops. Pipeline resulting loop | 134 | 134 | 134 | 134 | 9.39 | 1.26 | 1.26 | 5.10 | 5.10 |
| | Merge loops. Pipeline resulting loop. Replace pow | 117 | 117 | 117 | 117 | 7.26 | 0.85 | 0.85 | 7.56 | **7.56** |
| Standard Deviation | No directives (O1 and O2) | 11 | 11 | 11 | 11 | 8.13 | 0.09 | 0.09 | - | - |
| Signal Magnitude | No directives | 1789 | 1789 | 1789 | 1789 | 9.39 | 16.79 | 16.79 | - | - |
| | Pipeline loops | 225 | 225 | 225 | 225 | 9.39 | 2.11 | 2.11 | 7.95 | 7.95 |
| | Pipeline loops. Replace pow (O1) | 174 | 174 | 174 | 174 | 8.13 | 1.41 | 1.41 | 11.87 | 11.87 |
| | Pipeline loops. Unroll loops (f=2) | 162 | 162 | 162 | 162 | 9.39 | 1.52 | 1.52 | 11.04 | 11.04 |
| | Pipeline loops. Unroll loops (f=2) Replace pow | 111 | 111 | 111 | 111 | 8.13 | 0.90 | 0.90 | 18.61 | 18.61 |
| | Unroll loops | 51 | 51 | 51 | 51 | 9.80 | 0.50 | 0.50 | 33.58 | 33.58 |
| | Unroll loops. Replace pow (O2) | 35 | 35 | 35 | 35 | 9.20 | 0.32 | 0.32 | 52.18 | **52.18** |
| | Dataflow function. Pipeline loops | 150 | 150 | 42 | 42 | 9.39 | 1.41 | 1.41 | 11.93 | 11.93 |
| | Dataflow function. Pipeline loops. Replace pow | 132 | 132 | 34 | 34 | 8.13 | 1.07 | 1.07 | 15.65 | 15.65 |
| | Dataflow function. Unroll loops | 61 | 61 | 31 | 31 | 9.39 | 0.57 | 0.57 | 29.33 | 29.33 |
| | Dataflow function. Unroll loops. Replace pow | 44 | 44 | 14 | 14 | 8.13 | 0.36 | 0.36 | 46.95 | 46.95 |
| | Merge loops. Pipeline resulting loop | 62 | 62 | 62 | 62 | 9.39 | 0.58 | 0.58 | 28.85 | 28.85 |
| | Merge loops. Pipeline resulting loop. Replace pow | 46 | 46 | 46 | 46 | 8.13 | 0.37 | 0.37 | 44.91 | 44.91 |

Table B.2: High level synthesis performance estimates for the *classifier* stage and the stage with feature extraction grouped with floats

| **Stage** | **Notes** | **Latency (Clock cycles)** | | **Interval (Clock cycles)** | | **Clock Period (ms)** | **Execution Time (ms)** | | **Speedup** | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | **Min** | **Max** | **Min** | **Max** | | **Min** | **Max** | **Min** | **Max** |
| Classifier | No directives | 18775 | 238839 | 18775 | 238839 | 9.63 | 180.88 | 2300.97 | - | - |
| | Pipeline loops | 46631 | 46631 | 46631 | 46631 | 16.28 | 759.20 | 759.20 | 0.24 | 3.03 |
| | Pipeline loops. Replace pow | 37638 | 37638 | 37638 | 37638 | 15.21 | 572.47 | 572.47 | 0.32 | 4.02 |
| | Pipeline loops. Unroll loops (f=2) | 51367 | 51367 | 51367 | 51367 | 16.93 | 869.64 | 869.64 | 0.21 | 2.65 |
| | Pipeline loops. Unroll loops (f=2) Replace pow | 42903 | 42903 | 42903 | 42903 | 16.93 | 726.35 | 726.35 | 0.25 | 3.17 |
| | Unroll loops | 47667 | 47667 | 47667 | 47667 | 9.63 | 459.22 | 459.22 | 0.39 | 5.01 |
| | Unroll loops. Replace pow | 38674 | 38674 | 38674 | 38674 | 9.63 | 372.59 | 372.59 | 0.49 | 6.18 |
| | Pipeline nested loop. Unroll rest | 3315 | 3315 | 3315 | 3315 | 9.63 | 31.94 | 31.94 | 5.66 | 72.05 |
| | Pipeline nested. loop. Unroll rest. Replace pow (O2) | 3298 | 3298 | 3298 | 3298 | 9.63 | 31.77 | 31.77 | 5.69 | **72.42** |
| | O1 | 37101 | 37101 | 37101 | 37101 | 15.21 | 564.31 | 564.31 | 0.32 | 4.08 |
| Feature Extraction Grouped | No directives | 3745 | 3745 | 3745 | 3745 | 9.39 | 35.15 | 35.15 | - | - |
| | Best combination | 693 | 693 | 693 | 693 | 9.80 | 6.79 | 6.79 | 5.17 | 5.17 |
| | Best combination. Replace pow | 626 | 626 | 626 | 626 | 9.20 | 5.76 | 5.76 | 6.11 | 6.11 |
| | Best combination. Merging functions. Replace pow | 284 | 284 | 284 | 284 | 9.90 | 2.81 | 2.81 | 12.50 | **12.50** |

Table B.3: Initiation intervals obtained for all the versions that used loop pipelining with floats

| **Stage** | **Notes** | **Initiation Intervals** |
|---|---|---|
| Mean | Pipeline loops (O1) | 5 |
| | Pipeline loops. Unroll loops (f=2) | 8 |
| Variance | Pipeline loops | 1, 1, 5 |
| | Pipeline loops. Replace pow (O1) | 1, 1, 5 |
| | Pipeline loops. Unroll loops (f=2) | 1, 1, 8 |
| | Pipeline loops. Unroll loops (f=2). Replace pow | 1, 1, 8 |
| | Dataflow Function. Pipeline loops | 1, 1, 5 |
| | Dataflow Function. Pipeline loops. Replace pow | 1, 1, 5 |
| | Merge loops. Pipeline resulting loop | 5 |
| | Merge loops. Pipeline resulting loop. Replace pow | 5 |
| Signal Magnitude | Pipeline loops | 1, 1, 1, 1, 1, 1, 1 |
| | Pipeline loops. Replace pow (O1) | 1, 1, 1, 1, 1, 1, 1 |
| | Pipeline loops. Unroll loops (f=2) | 1, 1, 1, 1, 1, 1, 1 |
| | Pipeline loops. Unroll loops (f=2). Replace pow | 1, 1, 1, 1, 1, 1, 1 |
| | Dataflow Function. Pipeline loops | 1, 1, 1, 1, 1, 1, 1 |
| | Dataflow Function. Pipeline loops. Replace pow | 1, 1, 1, 1, 1, 1, 1 |
| | Merge loops. Pipeline resulting loop | 1 |
| | Merge loops. Pipeline resulting loop. Replace pow | 1 |
| Classifier | Pipeline loops | 1, 4, 1, 1, 2, 1 |
| | Pipeline loops. Replace pow | 1, 4, 1, 1, 2, 1 |
| | Pipeline loops. Unroll loops (f=2) | 1, 8, 1, 1, 4, 1 |
| | Pipeline loops. Unroll loops (f=2). Replace pow | 1, 8, 1, 1, 4, 1 |
| | Pipeline nested loop. Unroll rest | 6 |
| | Pipeline nested loop. Unroll rest. Replace pow (O2) | 6 |
| | O1 | 4, 1, 1 |

Table B.4: High level synthesis resource usage estimates for all the stages with floats

| **Stage** | **Notes** | **BRAM (%)** | **DSP (%)** | **FF (%)** | **LUT (%)** |
|---|---|---|---|---|---|
| Mean | No directives | 0.00 | 0.91 | 1.00 | 2.90 |
| | Pipeline loops (O1) | 0.00 | 0.91 | 1.00 | 2.93 |
| | Pipeline loops. Unroll loops (f=2) | 0.00 | 0.91 | 1.03 | 3.03 |
| | Unroll loops (O2) | 0.00 | 0.91 | 1.07 | 3.73 |
| Variance | No directives | 4.64 | 7.27 | 4.05 | 8.48 |
| | Pipeline loops | 4.64 | 7.27 | 4.29 | 8.64 |
| | Pipeline loops. Replace pow (O1) | 0.00 | 2.27 | 1.59 | 4.19 |
| | Pipeline loops. Unroll loops (f = 2) | 10.71 | 14.55 | 7.40 | 14.66 |
| | Pipeline loops. Unroll loops (f = 2). Replace pow | 1.43 | 4.55 | 2.02 | 5.76 |
| | Unroll loops | 4.64 | 8.18 | 4.13 | 9.85 |
| | Unroll loops. Replace pow (O2) | 0.00 | 2.27 | 1.27 | 4.51 |
| | Dataflow function. Pipeline loops | 5.36 | 8.18 | 4.60 | 9.44 |
| | Dataflow function. Pipeline loops. Replace pow | 0.00 | 3.18 | 1.91 | 4.97 |
| | Dataflow function. Unroll loops | 96.43 | **131.82** | 56.34 | **113.19** |
| | Dataflow function. Unroll loops. Replace pow | 0.00 | 41.82 | 8.80 | 32.66 |
| | Merge loops. Pipeline resulting loop | 4.64 | 7.27 | 3.91 | 8.13 |
| | Merge loops. Pipeline resulting loop. Replace pow | 0.00 | 2.27 | 1.26 | 3.67 |
| Standard Deviation | No directives (O1 and O2) | 0.00 | 0.00 | 0.39 | 1.26 |
| Signal Magnitude | No directives | 4.64 | 7.27 | 4.15 | 8.61 |
| | Pipeline loops | 4.64 | 7.27 | 4.86 | 8.90 |
| | Pipeline loops. Replace pow (O1) | 0.00 | 2.27 | 2.13 | 4.49 |
| | Pipeline loops. Unroll loops (f = 2) | 13.57 | 14.55 | 8.49 | 16.44 |
| | Pipeline loops. Unroll loops (f = 2). Replace pow | 4.29 | 4.55 | 3.08 | 7.61 |
| | Unroll loops | 27.86 | 41.82 | 19.73 | 37.54 |
| | Unroll loops. Replace pow (O2) | 0.00 | 11.82 | 3.82 | 10.75 |
| | Dataflow function. Pipeline loops | 16.07 | 20.91 | 10.71 | 20.02 |
| | Dataflow function. Pipeline loops. Replace pow | 0.00 | 5.91 | 2.64 | 6.62 |
| | Dataflow function. Unroll loops | **289.29** | **376.36** | **169.61** | **336.60** |
| | Dataflow function. Unroll loops. Replace pow | 0.00 | **106.36** | 25.87 | 100.00 |
| | Merge loops. Pipeline resulting loop | 13.93 | 20.91 | 9.48 | 18.17 |
| | Merge loops. Pipeline resulting loop. Replace pow | 0.00 | 5.91 | 1.51 | 4.70 |
| Classifier | No directives | 4.64 | 7.73 | 4.22 | 11.23 |
| | Pipeline loops | 4.64 | 7.73 | 4.98 | 12.44 |
| | Pipeline loops. Replace pow | 0.00 | 2.73 | 2.03 | 8.14 |
| | Pipeline loops. Unroll loops (f = 2) | 5.36 | 8.18 | 6.32 | 16.80 |
| | Pipeline loops. Unroll loops (f = 2). Replace pow | 0.71 | 3.18 | 3.58 | 12.55 |
| | Unroll loops | 5.36 | 9.55 | 6.01 | 23.29 |
| | Unroll loops. Replace pow | 0.71 | 4.55 | 3.31 | 18.97 |
| | Pipeline nested loop. Unroll other loops | 10.00 | 17.73 | 10.37 | 29.90 |
| | Pipeline nested loop. Unroll other loops. Replace pow (O2) | 0.71 | 7.73 | 5.25 | 21.58 |
| | O1 | 0.00 | 3.64 | 1.98 | 10.05 |
| Feature Extraction Grouped | No directives | 13.93 | 23.64 | 13.04 | 32.33 |
| | Best combination of functions | 37.86 | 58.18 | 30.27 | 61.91 |
| | Best combination of functions. Replace pow | 0.71 | 18.18 | 9.04 | 26.20 |
| | Best combination. Merging functions. Replace pow | 0.71 | 15.00 | 7.14 | 22.79 |

Table B.5: High level synthesis performance estimates for the *mean*, *variance*, *standardDeviation* and *signalMagnitude* stages with doubles

| **Stage** | **Notes** | **Latency (Clock cycles)** | | **Interval (Clock cycles)** | | **Clock Period (ms)** | **Execution Time (ms)** | | **Speedup** | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | **Min** | **Max** | **Min** | **Max** | | **Min** | **Max** | **Min** | **Max** |
| Mean | No directives | 157 | 157 | 157 | 157 | 8.62 | 1.35 | 1.35 | - | - |
| | Pipeline loops (O1) | 123 | 123 | 123 | 123 | 8.62 | 1.06 | 1.06 | 1.28 | **1.28** |
| | Pipeline loops. Unroll loops (f=2) | 106 | 106 | 106 | 106 | 16.46 | 1.75 | 1.75 | 0.78 | 0.78 |
| | Unroll loops (O2) | 122 | 122 | 122 | 122 | 10.36 | 1.26 | 1.26 | 1.07 | 1.07 |
| Variance | No directives | 1545 | 1545 | 1545 | 1545 | 9.51 | 14.70 | 14.70 | - | - |
| | Pipeline loops | 236 | 236 | 236 | 236 | 9.51 | 2.25 | 2.25 | 6.55 | 6.55 |
| | Pipeline loops. Replace pow (O1) | 176 | 176 | 176 | 176 | 8.62 | 1.52 | 1.52 | 9.69 | 9.69 |
| | Pipeline loops. Unroll loops (f=2) | 201 | 201 | 201 | 201 | 16.46 | 3.31 | 3.31 | 4.44 | 4.44 |
| | Pipeline loops. Unroll loops (f=2) Replace pow | 141 | 141 | 141 | 141 | 16.46 | 2.32 | 2.32 | 6.33 | 6.33 |
| | Unroll loops | 194 | 194 | 194 | 194 | 10.87 | 2.11 | 2.11 | 6.97 | 6.97 |
| | Unroll loops. Replace pow (O2) | 133 | 133 | 133 | 133 | 10.47 | 1.39 | 1.39 | 10.56 | 10.56 |
| | Dataflow function. Pipeline loops | 239 | 239 | 124 | 124 | 9.51 | 2.27 | 2.27 | 6.46 | 6.46 |
| | Dataflow function. Pipeline loops. Replace pow | 178 | 178 | 124 | 124 | 8.62 | 1.53 | 1.53 | 9.58 | 9.58 |
| | Dataflow function. Unroll loops | 203 | 203 | 121 | 121 | 10.36 | 2.10 | 2.10 | 6.99 | 6.99 |
| | Dataflow function. Unroll loops. Replace pow | 141 | 141 | 121 | 121 | 10.36 | 1.46 | 1.46 | 10.07 | 10.07 |
| | Merge loops. Pipeline resulting loop | 195 | 195 | 195 | 195 | 9.51 | 1.86 | 1.86 | 7.92 | 7.92 |
| | Merge loops. Pipeline resulting loop. Replace pow | 134 | 134 | 134 | 134 | 8.62 | 1.16 | 1.16 | 12.72 | **12.72** |
| Standard Deviation | No directives (O1 and O2) | 30 | 30 | 30 | 30 | 8.62 | 0.26 | 0.26 | - | - |
| Signal Magnitude | No directives | 4669 | 4669 | 4669 | 4669 | 9.51 | 44.42 | 44.42 | - | - |
| | Pipeline loops | 385 | 385 | 385 | 385 | 9.51 | 3.66 | 3.66 | 12.13 | 12.13 |
| | Pipeline loops. Replace pow (O1) | 205 | 205 | 205 | 205 | 8.62 | 1.77 | 1.77 | 25.13 | 25.13 |
| | Pipeline loops. Unroll loops (f=2) | 322 | 322 | 322 | 322 | 9.51 | 3.06 | 3.06 | 14.50 | 14.50 |
| | Pipeline loops. Unroll loops (f=2) Replace pow | 142 | 142 | 142 | 142 | 8.62 | 1.22 | 1.22 | 36.28 | 36.28 |
| | Unroll loops | 118 | 118 | 118 | 118 | 9.92 | 1.17 | 1.17 | 37.94 | 37.94 |
| | Unroll loops. Replace pow (O2) | 57 | 57 | 57 | 57 | 9.40 | 0.54 | 0.54 | 82.88 | **82.88** |
| | Dataflow function. Pipeline loops | 216 | 216 | 89 | 89 | 9.51 | 2.06 | 2.06 | 21.62 | 21.62 |
| | Dataflow function. Pipeline loops. Replace pow | 155 | 155 | 53 | 53 | 8.62 | 1.34 | 1.34 | 33.24 | 33.24 |
| | Dataflow function. Unroll loops | 128 | 128 | 78 | 78 | 9.51 | 1.22 | 1.22 | 36.48 | 36.48 |
| | Dataflow function. Unroll loops. Replace pow | 66 | 66 | 31 | 31 | 8.62 | 0.57 | 0.57 | 78.05 | 78.05 |
| | Merge loops. Pipeline resulting loop | 129 | 129 | 129 | 129 | 9.51 | 1.23 | 1.23 | 36.19 | 36.19 |
| | Merge loops. Pipeline resulting loop. Replace pow | 68 | 68 | 68 | 68 | 8.62 | 0.59 | 0.59 | 75.76 | 75.76 |

Table B.6: High level synthesis performance estimates for the *classifier* stage and the stage with feature extraction grouped with doubles

| Stage | Notes | Latency (Clock cycles) | | Interval (Clock cycles) | | Clock Period (ms) | Execution Time (ms) | | Speedup | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Min | Max | Min | Max | | Min | Max | Min | Max |
| Classifier | No directives | 19330 | 555736 | 19330 | 555736 | 9.63 | 186.23 | 5353.96 | - | - |
| | Pipeline loops | 72024 | 72024 | 72024 | 72024 | 18.23 | 1313.21 | 1313.21 | 0.14 | 4.08 |
| | Pipeline loops. Replace pow | 39226 | 39226 | 39226 | 39226 | 17.95 | 703.91 | 703.91 | 0.26 | 7.61 |
| | Pipeline loops. Unroll loops (f=2) | 76759 | 76759 | 76759 | 76759 | 18.57 | 1425.26 | 1425.26 | 0.13 | 3.76 |
| | Pipeline loops. Unroll loops (f=2) Replace pow | 43432 | 43432 | 43432 | 43432 | 18.57 | 806.45 | 806.45 | 0.23 | 6.64 |
| | Unroll loops | 72019 | 72019 | 72019 | 72019 | 10.36 | 745.76 | 745.76 | 0.25 | 7.18 |
| | Unroll loops. Replace pow | 39750 | 39750 | 39750 | 39750 | 10.36 | 411.61 | 411.61 | 0.45 | 13.01 |
| | Pipeline nested loop. Unroll rest | 3379 | 3379 | 3379 | 3379 | 9.63 | 32.55 | 32.55 | 5.72 | 164.47 |
| | Pipeline nested. loop. Unroll rest. Replace pow (O2) | 3318 | 3318 | 3318 | 3318 | 9.63 | 31.97 | 31.97 | 5.83 | **167.49** |
| | O1 | 38689 | 38689 | 38689 | 38689 | 17.95 | 694.27 | 694.27 | 0.27 | 7.71 |
| Feature Extraction Grouped | No directives | 9366 | 9366 | 9366 | 9366 | 9.51 | 89.11 | 89.11 | - | - |
| | Best combination | 992 | 992 | 992 | 992 | 9.92 | 9.84 | 9.84 | 9.05 | 9.05 |
| | Best combination. Replace pow | 748 | 748 | 748 | 748 | 9.40 | 7.03 | 7.03 | 12.67 | 12.67 |
| | Best combination. Merging functions. Replace pow | 381 | 381 | 381 | 381 | 9.77 | 3.72 | 3.72 | 23.95 | **23.95** |

Table B.7: Initiation intervals obtained for all the versions that used loop pipelining with doubles

| Stage | Notes | Initiation Intervals |
|---|---|---|
| Mean | Pipeline loops (O1) | 5 |
| | Pipeline loops. Unroll loops (f=2) | 8 |
| Variance | Pipeline loops | 1, 1, 5 |
| | Pipeline loops. Replace pow (O1) | 1, 1, 5 |
| | Pipeline loops. Unroll loops (f=2) | 1, 1, 8 |
| | Pipeline loops. Unroll loops (f=2). Replace pow | 1, 1, 8 |
| | Dataflow Function. Pipeline loops | 1, 1, 5 |
| | Dataflow Function. Pipeline loops. Replace pow | 1, 1, 5 |
| | Merge loops. Pipeline resulting loop | 5 |
| | Merge loops. Pipeline resulting loop. Replace pow | 5 |
| Signal Magnitude | Pipeline loops | 1, 1, 1, 1, 1, 1, 1 |
| | Pipeline loops. Replace pow (O1) | 1, 1, 1, 1, 1, 1, 1 |
| | Pipeline loops. Unroll loops (f=2) | 1, 1, 1, 1, 1, 1, 1 |
| | Pipeline loops. Unroll loops (f=2). Replace pow | 1, 1, 1, 1, 1, 1, 1 |
| | Dataflow Function. Pipeline loops | 1, 1, 1, 1, 1, 1, 1 |
| | Dataflow Function. Pipeline loops. Replace pow | 1, 1, 1, 1, 1, 1, 1 |
| | Merge loops. Pipeline resulting loop | 1 |
| | Merge loops. Pipeline resulting loop. Replace pow | 1 |
| Classifier | Pipeline loops | 1, 4, 1, 1, 2, 1 |
| | Pipeline loops. Replace pow | 1, 4, 1, 1, 2, 1 |
| | Pipeline loops. Unroll loops (f=2) | 1, 8, 1, 1, 4, 1 |
| | Pipeline loops. Unroll loops (f=2). Replace pow | 1, 8, 1, 1, 4, 1 |
| | Pipeline nested loop. Unroll rest | 6 |
| | Pipeline nested loop. Unroll rest. Replace pow (O2) | 6 |
| | O1 | 4, 1, 1 |

Table B.8: High level synthesis resource usage estimates for all the stages with doubles

| Stage | Notes | BRAM (%) | DSP (%) | FF (%) | LUT (%) |
|---|---|---|---|---|---|
| Mean | No directives | 0.00 | 1.36 | 3.60 | 9.44 |
| | Pipeline loops (O1) | 0.00 | 1.36 | 3.60 | 9.47 |
| | Pipeline loops. Unroll loops (f=2) | 0.00 | 1.36 | 3.67 | 9.58 |
| | Unroll loops (O2) | 0.00 | 1.36 | 3.67 | 10.27 |
| Variance | No directives | 25.36 | 33.18 | 18.10 | 25.80 |
| | Pipeline loops | 25.36 | 33.18 | 18.34 | 25.74 |
| | Pipeline loops. Replace pow (O1) | 0.00 | 6.36 | 4.42 | 11.19 |
| | Pipeline loops. Unroll loops (f = 2) | 50.71 | 66.36 | 33.35 | 43.72 |
| | Pipeline loops. Unroll loops (f = 2). Replace pow | 2.86 | 12.73 | 5.57 | 14.70 |
| | Unroll loops | 23.93 | 33.18 | 18.08 | 26.41 |
| | Unroll loops. Replace pow (O2) | 0.00 | 6.36 | 4.16 | 11.63 |
| | Dataflow function. Pipeline loops | 26.07 | 34.55 | 18.94 | 27.55 |
| | Dataflow function. Pipeline loops. Replace pow | 2.14 | 7.73 | 4.91 | 13.32 |
| | Dataflow function. Unroll loops | **430.71** | **598.64** | **272.66** | **328.69** |
| | Dataflow function. Unroll loops. Replace pow | 0.00 | **155.91** | 19.97 | 75.74 |
| | Merge loops. Pipeline resulting loop | 23.93 | 33.18 | 18.01 | 25.08 |
| | Merge loops. Pipeline resulting loop. Replace pow | 0.00 | 6.36 | 4.08 | 10.70 |
| Standard Deviation | No directives (O1 and O2) | 0.00 | 0.00 | 1.75 | 4.37 |
| Signal Magnitude | No directives | 28.21 | 33.18 | 17.30 | 24.68 |
| | Pipeline loops | 28.21 | 33.18 | 18.01 | 24.13 |
| | Pipeline loops. Replace pow (O1) | 4.29 | 6.36 | 3.98 | 9.24 |
| | Pipeline loops. Unroll loops (f = 2) | 56.43 | 66.36 | 35.36 | 46.82 |
| | Pipeline loops. Unroll loops (f = 2). Replace pow | 8.57 | 12.73 | 7.47 | 17.31 |
| | Unroll loops | **143.57** | **196.36** | 93.78 | **112.12** |
| | Unroll loops. Replace pow (O2) | 0.00 | 35.45 | 12.01 | 26.18 |
| | Dataflow function. Pipeline loops | 76.07 | 98.18 | 73.30 | 56.44 |
| | Dataflow function. Pipeline loops. Replace pow | 4.29 | 17.73 | 5.20 | 13.76 |
| | Dataflow function. Unroll loops | **1292.14** | **1767.27** | **830.69** | **990.98** |
| | Dataflow function. Unroll loops. Replace pow | 0.00 | **319.09** | 71.50 | **231.57** |
| | Merge loops. Pipeline resulting loop | 71.79 | 98.18 | 45.99 | 55.42 |
| | Merge loops. Pipeline resulting loop. Replace pow | 0.00 | 17.73 | 4.17 | 12.06 |
| Classifier | No directives | 23.93 | 33.64 | 16.53 | 25.19 |
| | Pipeline loops | 23.93 | 33.64 | 18.33 | 26.20 |
| | Pipeline loops. Replace pow | 0.00 | 6.82 | 3.25 | 12.31 |
| | Pipeline loops. Unroll loops (f = 2) | 24.64 | 34.09 | 19.65 | 32.32 |
| | Pipeline loops. Unroll loops (f = 2). Replace pow | 0.71 | 7.27 | 5.30 | 18.43 |
| | Unroll loops | 24.64 | 35.91 | 19.29 | 42.99 |
| | Unroll loops. Replace pow | 0.71 | 9.09 | 5.37 | 28.70 |
| | Pipeline nested loop. Unroll other loops | 55.71 | 70.45 | 35.40 | 61.66 |
| | Pipeline nested loop. Unroll other loops. Replace pow (O2) | 7.86 | 16.82 | 8.15 | 34.05 |
| | O1 | 0.00 | 7.73 | 3.17 | 15.97 |
| Feature Extraction Grouped | No directives | 76.79 | **102.27** | 64.78 | **103.82** |
| | Best combination of functions | **192.86** | **265.45** | **141.29** | **190.68** |
| | Best combination of functions. Replace pow | 1.43 | 50.91 | 29.88 | 71.87 |
| | Best combination. Merging functions. Replace pow | 1.43 | 43.18 | 24.52 | 55.86 |