

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO



Runtime Management of Heterogeneous Compute Resources in Embedded Systems

Luís Miguel Sousa

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Nuno Paulino

Second Supervisor: João Canas Ferreira

October 29, 2021

Resumo

Os avanços em tecnologia de semicondutores já não ocorrem ao ritmo a que a indústria se tinha acostumado. Estamos agora a viver aquilo que é considerado por muitos a era pós-Moore. De maneira a se conseguir continuar a escalar a capacidade de processamento, arquitecturas cada vez mais heterogéneas têm sido desenvolvidas e tem-se assistido a uma proliferação de aceleradores de uso específico.

Neste documento apresentamos uma *framework* de compilação de código-fonte para código-fonte capaz de transformar programas comuns escritos em C/C++ em programas adequados a arquitecturas heterogéneas. Isto é conseguido ao reaproveitar o código original para gerar aceleradores por recurso a HLS mantendo a capacidade original de execução em CPU, inserindo código especializado para a gestão de memória e tarefas e injectando um algoritmo de escalonamento definido pelo utilizador. Propomos ainda um tal algoritmo de escalonamento baseado numa implementação flexível em C/C++ para HLS de uma Árvore de Hoeffding. Os parâmetros de implementação da árvore incluem a quantidade de atributos contidos em cada conjunto de dados, o número de classificações possíveis e o número máximo de nós que a árvore pode conter.

Com recurso a uma placa MPSoC ZCU102 da Xilinx, avaliamos o desempenho da nossa implementação da Árvore de Hoeffding a classificar vários conjuntos de dados. Estes variam no número de exemplos contidos (N), número de classes de saída (K) e número de atributos dos dados (D). Comparamos execuções puramente de software no CPU ARM Cortex-A53 da placa com implementações em hardware dos algoritmos da árvore com recurso à ferramenta Vitis HLS da Xilinx. Avaliamos ainda qual a utilização de recursos e frequência de relógio para as várias implementações em hardware quando são utilizados diferentes números de atributos e números de classes de saída. Para problemas de complexidade D3, K5, N40000, a implementação por HLS de uma só árvore de decisão opera a 103MHz e é capaz de executar tarefas de inferência 8.3× mais rápido do que um CPU ARM Cortex-A53 a 1.2 GHz. Demonstramos a capacidade da *framework Tribble* de diminuir o custo de desenvolvimento para arquitecturas heterogéneas ao validar o fluxo de compilação proposto e que um algoritmo de escalonamento baseado numa Árvore de Hoeffding é capaz de atingir uma precisão de 96% em conjuntos de simulações de kernels hipotéticos.

Abstract

Advancements in semiconductor technology do not currently occur at the pace the industry had been accustomed to. We have entered what is considered by many to be the post-Moore era. In order to continue scaling performance, increasingly heterogeneous architectures are being developed and the use of special purpose accelerators is on the rise.

We present *Tribble*, a source-to-source framework capable of transforming regular C/C++ programs to execute on heterogeneous architectures. This is done by retargeting kernel source code to an HLS compiler while keeping the original capability of CPU execution, inserting custom code for task and memory management and injecting a user-defined scheduler algorithm. We also propose a runtime scheduling algorithm based on a flexible C/C++ HLS implementation of the Hoeffding Tree. The implementation parameters of the tree include the number of attributes of the samples under classification, the number of output classes, and the maximum number of nodes to which the tree is allowed to grow.

Targeting a Xilinx MPSoC ZCU102 device, we evaluate the classification performance of our Hoeffding Tree implementation on several datasets of varying sample sizes (N), number of output classes (K) and number of sample attributes (D). We compare software-only execution on the embedded ARM Cortex-A53 core, with hardware implementations of the tree algorithm using Xilinx's Vitis HLS tool. For the hardware implementations, we also evaluated the resource consumption and clock frequency for different numbers of classes and attributes. For a problem size of D3, K5, N40000, the HLS implementation of a single decision tree operating at 103MHz is capable of 8.3× faster inference than the 1.2 GHz ARM Cortex-A53 core. We demonstrate the capabilities of *Tribble* for easing the burden of developing code for heterogeneous architectures by validating the full-flow compilation capability and also that the proposed Hoeffding Tree-based scheduler is able to learn datasets based on hypothetical kernels' computational loads with an accuracy of 96%.

Acknowledgements

I would like to thank my supervisors, Nuno Paulino and João Canas Ferreira, for their support and discussion on the multitude of subjects surrounding this dissertation.

I wish to convey my thanks to Fábio Gaspar and João Bispo whose informal job as code ducks and contributions to the Clava tool were instrumental for the completion of this work and to my friends - especially those at the IEEE University of Porto Student Branch.

In no means comparable is the amount of appreciation that I have for the continued support given to me by my parents throughout my life. Without it I would not be the same person and would not have been able to accomplish what I did.

Luís Miguel Sousa

*“It’s hardware that makes a machine fast.
It’s software that makes a fast machine slow.”*

Craig Bruce

Contents

1	Introduction	1
1.1	Context	1
1.1.1	High Level Synthesis	2
1.1.2	Dynamic Partial Reconfiguration	3
1.2	Problem	3
1.3	Objective	4
1.4	Document Structure	4
2	Background	5
2.1	Field Programmable Gate Arrays	5
2.1.1	Benefits of special-purpose circuitry	6
2.2	High-Level Synthesis	8
3	Related Work	10
3.1	Reconfigurable Hardware Resource Management Frameworks	10
3.1.1	SkePU	10
3.1.2	Offload Annotations	11
3.1.3	Runtime Decisions	12
3.1.4	QBC	12
3.1.5	HIS	13
3.1.6	Digest	13
3.2	Decision Trees	14
3.2.1	VFDT	15
3.2.2	Gaussian Method	15
3.2.3	Quantile Estimation	15
4	General Approach	16
4.1	Scope	16
4.2	Proposed Architecture	17
4.2.1	Code Injection	17
4.2.2	Scheduling	18
5	<i>Tribble</i> Source-to-Source Framework	19
5.1	Folder Structure	20
5.2	Kernel Pragmas	21
5.3	Enforcing kernel <code>void</code> type	21
5.4	Estimating Kernel Compute Load	22
5.5	Scheduling	25

5.6	Profiling and Training	26
5.7	OpenCL Kernel Arguments	27
5.8	Vitis Incompatibilities	29
5.9	<code>main()</code> Template	29
5.10	<i>HLSAnalysis</i>	31
5.11	<i>Trials and Tribble-ations</i>	31
5.12	Toolchain	34
5.12.1	Project Creation	34
5.12.2	<i>Tribble</i> output integration	35
6	Hardware Resource Scheduler	36
6.1	HLS Hoeffding Tree Implementation	36
6.1.1	NodeData	38
6.1.2	Node	38
6.1.3	BinaryTree and HoeffdingTree	38
6.1.4	TypeChooser and TypeChooserMath	38
6.2	Tree Visualisation	39
6.3	Limitations	39
7	Experimental Evaluation	40
7.1	Experimental Setup	40
7.2	Validation Threats	40
7.3	Evaluation	41
7.3.1	Hoeffding Tree Throughput	41
7.3.2	Hoeffding Tree Resource Utilisation	42
7.3.3	Hoeffding Tree Kernel Performance	45
7.3.4	<i>Tribble</i> Code Transformations	45
7.3.5	Execution of Generated Code	46
7.3.6	Hoeffding Tree - Learning from Operation Counts	46
7.3.7	Hoeffding Tree Scheduler	53
8	Conclusions	55
8.1	Contributions	56
8.2	Future Work	56
A	Publications	58
	LATTE'21	59
	ISCAS 2022	62
B	<i>adi</i> Code Transformations	66
B.1	Original Code	66
B.1.1	CxxTemplates/OCLH_main.cpp	66
B.1.2	CxxSource/adi.cpp	67
B.1.3	CxxSource/adi.h	70
B.2	<i>Tribble</i> Output	72
B.2.1	woven_code/CxxTemplates/OCLH_main.cpp	73
B.2.2	woven_code/CxxSource/adi.cpp	73
B.2.3	woven_code/CxxSource/adi.h	78
B.2.4	woven_code/CxxSource/kernels.cpp	78

C	<i>adi</i> Kernel Optimisations	81
D	Dataset Creation Code	84
	References	90

List of Figures

2.1	Task Parallelism within a Run. Source: Vitis Unified Software Development Platform 2021.1 Documentation.	6
2.2	Task Pipelining. Source: Vitis Unified Software Development Platform 2021.1 Documentation.	7
2.3	Task Parallelism with Pipelining. Source: Vitis Unified Software Development Platform 2021.1 Documentation.	7
2.4	Task Parallelism and Pipelining within a Run, Pipelining of Runs, and Pipelining within a Task. Source: Vitis Unified Software Development Platform 2021.1 Documentation.	7
2.5	C/C++ to RTL synthesis flow in <i>Xilinx Vitis HLS</i> . It consists of 2 main components – 1. Front-end: This component parses the code expressed in C/C++ or OpenCL, applies front-end and middle-end transformations using the Clang/LLVM tool chain. 2. Back-end: This phase takes an LLVM IR input and performs FPGA-specific lowering and scheduling until the final step, RTL generation. Source: Xilinx Blog.	8
4.1	Diagram of internal architecture of Xilinx UltraScale and UltraScale+ family of integrated circuit devices. A processing system portion implemented as hard logic interfaces with the re-configurable area portion where accelerators are hosted. Source: Xilinx product pages.	16
4.2	Diagram describing the proposed workflow.	17
5.1	XRT support in Vitis platform	35
6.1	Hoeffding Tree kernel generation and workflow diagram.	37
7.1	Visualisation of a 3 dimensional (D=3), 5 cluster dataset (K=5), with 40k points (N=40.000). The number of points in each cluster is random.	42
7.2	Two dimensional projection over the Z-axis of how the dataset from figure 7.1 was classified by a tree.	43
7.3	Evolution of the size of Tree objects in bytes by changing Nd, D and K. Each bar in every grouping, depicts a tree with a max number of nodes from 2^0 to 2^7 . Datatype is <code>float</code>	43
7.4	Evaluation of <i>adi</i> kernel runtimes on the the CPU and FPGA portions of the ZCU102 board using 'double' as the kernel datatype. $tsteps = N/2$. $N \in [20; 146]$	47
7.5	Comparison between the FPGA and CPU execution times for the <i>adi</i> kernel using 'double' as the datatype. $tsteps = N/2$. $N \in [20; 146]$	47
7.6	Evaluation of <i>adi</i> kernel runtimes on the the CPU and FPGA portions of the ZCU102 board using 'float' as the kernel datatype. $tsteps = N/2$. $N \in [50; 750]$	48

7.7	Comparison between the FPGA and CPU execution times for the <i>adi</i> kernel using 'float' as the datatype. $tsteps = N/2$. $N \in [50; 750]$	48
7.8	Evaluation of <i>adi</i> kernel runtimes on the the CPU and FPGA portions of the ZCU102 board using 'float' as the kernel datatype. $tsteps = N/2$. $N \in [20; 280]$. Kernel has had light manual optimisations made to the code.	49
7.9	Comparison between the FPGA and CPU execution times for the <i>adi</i> kernel using 'float' as the datatype. $tsteps = N/2$. $N \in [20; 280]$. Kernel has had light manual optimisations made to the code.	49
7.10	Visualisation of the decision tree models resulting from learning the dataset generated for a hypothetical <i>2mm</i> kernel. Leaf nodes attempt to split at every 200 samples. Sample counts on each non-leaf node represent the state of the node when it was split.	53

List of Tables

2.1	Full-adder logic table.	5
3.1	Digest of decision engine characteristics	13
7.1	Training and inference times for four synthetic clustering datasets, for the ARM CPU (1.2Ghz) and the FPGA (103MHz).	43
7.2	N, D, K and Nd effects on FPGA Resource Utilisation	44
7.3	Training time and Accuracy (Acc.) for Covertypes and Bank datasets, for the ARM CPU (1.2Ghz) and the FPGA (103MHz)	45
7.4	Dataset characteristics for hypothetical optimised kernels.	51
7.5	Hoeffding Tree accuracy results for hypothetical optimised kernels. Leaf nodes do split trials on every 200 samples collected.	52
7.6	Hoeffding Tree accuracy results for hypothetical optimised kernels. Leaf nodes do split trials on every 50 samples collected.	52
7.7	Hoeffding Tree accuracy results for hypothetical optimised kernels. Leaf nodes do split trials on every 10 samples collected.	52
7.8	Scheduler accuracy results hypothetical optimised kernels. Leaf nodes do split trials on every 50 samples collected.	54

Abbreviations and Symbols

AP	Arbitrary Precision
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
AST	Abstract Syntax Tree
BRAM	Block RAM
CPU	Central Processing Unit
DPR	Dynamic Partial Reconfiguration
DSL	Domain Specific Language
DSP	Digital Signal Processor
DT	Decision Tree
FF	Flip-Flop
FLOP	Floating Point Operation
FPGA	Field-Programmable-Gate-Array
FSM	Finite State Machines
GCC	GNU C Compiler
GNU	GNU is Not Unix
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDL	Hardware Description Language
HIS	Hybrid static-dynamic Implementation Selection
HLS	High-Level Synthesis
HwA	Hardware Accelerator
IDE	Integrated Development Environment
II	Initiation Interval
IR	Intermediate Representation
I/O	Input/Output
IOP	Integer Operation
JSON	JavaScript Object Notation
LLVM-IR	LLVM Intermediate Representation
LUT	Lookup Table

ML	Machine Learning
MPSoC	Multi Processor System on a Chip
OA	Offload Annotations
OS	Operating System
QBC	Query by Committee
RTL	Register-transfer Level
SCEV	ScalarEvolution
SoC	System on a Chip
VFDT	Very Fast Decision Tree
VHDL	VHSIC Hardware Description Language
XRT	Xilinx Runtime

Chapter 1

Introduction

1.1 Context

Over the years, with the decrease of transistor sizes, the industry has been able to scale transistor density and the other benefits that accompany a new silicon manufacturing node, resulting in a predictable increase in compute capacity of new chip architectures. However, with the ever more pressing issue arising from the phenomenon known as the *death* or *slowdown* of both Moore's Law and Dennard Scaling, compute performance has not been increasing at the rate the industry had become accustomed to over the decades [1]. Without the performance benefits provided by new manufacturing nodes in new chips, the industry started looking at other ways of achieving speedups. One of those ways has been the shift from a mostly homogeneous architecture to an increasingly heterogeneous one.

The execution performance of an application is a function of the quality of the software, and the computing power of the hardware. That is, an application's performance can be maximised if it is executed on specialised hardware, both from a time and power consumption perspective. Unsurprisingly, in the embedded domain, it is not uncommon for compute platforms to be heterogeneous [2, 3]. That is, to contain a main processor, and several specialised co-processors for specific functions (e.g., cryptography, image processing). It is up to the programmer to understand the underlying hardware platform, schedule workloads onto different components and to synchronise their behaviour. As customers demand higher performance and functionality, these systems tend to increase in complexity and therefore require more development time and expertise. Furthermore, since silicon area is expensive, the use of Hardware Accelerators (HwAs) is only justified under certain conditions. Firstly, that they are used frequently to justify not only the area they occupy but also the design time. Secondly, that their workload is both well defined, and amenable for parallelization, so that performance benefits can be maximised. For example, cryptography, video codecs, and neural networks [4], among others.

One of the platforms that can be used for this kind of approach are Field-Programmable-Gate-Arrays (FPGAs). Although these devices lack the advantages of full-custom, i.e. Application Specific Integrated Circuit (ASIC), implementations [5], they introduce new paradigms to these

heterogeneous systems. **FPGAs** allow the application to define custom circuitry to be implemented. This task-specific circuitry is more efficient at executing the task than a general-purpose Central Processing Unit (**CPU**).

The capability for reconfiguration allows for application-specific circuits to be deployed at small or medium scales without the need for **ASIC** fabrication, which is only viable at large scales. Instead, a single underlying chip can be used to implement the **HwAs** required by a specific application. Given the present trend towards edge computing e.g., distributed Machine Learning (**ML**) [6], the respective need for fast design of specialised computing devices, and the aforementioned cost benefits, **FPGAs** gain new relevance. Additionally, algorithms are subject to change due to performance reasons, different application needs, or bug fixes. As **FPGAs** are reconfigurable, the cost cutting is multiplied as no new devices need to be fabricated and deployed.

1.1.1 High Level Synthesis

Manual hardware design is a difficult process. It requires extensive knowledge of computer architecture, signal propagation, power management and many more technical domains, all of them taking decades to fully master. Due to this, development is mainly done by large specialised teams during several months to years.

FPGAs reduce the development burden by removing the complexity of physical design and by providing tools that allow developers to work using Hardware Description Languages (**HDLs**) that then map to predefined blocks on the **FPGA**. Vendors place great invest into these tools, automating the *mapping* of the algorithm onto the blocks and the subsequent *routing* of the connections between blocks. This process is known as *synthesis*.

Although much simpler, this process is still not what one could call "software developer friendly" as a detailed understanding of the underlying architecture is needed to produce any high-performance design. To that end, industry and academia have invested in a technique called *High Level Synthesis* [7, 8]. This takes high-level C, C++, Fortran and some times even Python code and converts it to an Register-transfer Level (**RTL**) description on a **HDL** like Verilog or VHDL.

Designing circuits using a higher-level language has several advantages. The move from circuit description to behaviour description lowers the barrier of entry but it also means that the overall code base is much smaller. Less code to write, review and maintain. Leveraging existing software development tools like linters, static analysers, debuggers and Integrated Development Environments (**IDEs**) means that the process itself is supported by a plethora of mature, easy-to-use and effective tools. Testing the code is much faster and late-stage changes and experimentation can also occur seamlessly. In **RTL** any major change or feature addition often requires a major redesign of the entire circuit. Using High-Level Synthesis (**HLS**) that translates into changing the code and re-running the compiler. Perhaps more importantly, as only behaviours are being coded, there is no tie to any architecture or manufacturing process. Changing any of these parameters is as simple as recompiling the code.

The economic benefit of this technology is clear. Much faster and flexible development that enables a significantly shorter time-to-market.

However, just as with HDL based designs, some expert knowledge is required to create performant HLS-based designs. Specifically, although low level hardware details are abstracted away, the expert knowledge required for efficiency is now centered around the computing model. Mainly, on dataflow parallelism, data partitioning, and data streaming.

1.1.2 Dynamic Partial Reconfiguration

A unique property of FPGAs is the capability of switching between accelerator circuits at run-time, which is referred to as Dynamic Partial Reconfiguration (DPR) [9]. This feature, which has not seen widespread adoption in real-world applications, allows for a targeted FPGA area to be reconfigured while the device is in operation without disturbing operations occurring in the rest of the compute fabric. This effectively modifies the circuit implemented in the delimited region, allowing for the same resources to implement multiple functions in a time-multiplexed manner. A set of accelerators can make use of the same FPGA region using this technique, which can be referred to as accelerator *hot-swapping* [10]. Allowing an application to hot-swap accelerators instead of solely dealing with a fixed, predefined set, implemented at boot-time, greatly expands on the possibilities of using FPGA devices as platforms for implementing accelerators. Accelerators that are not used concurrently can be swapped out as needed freeing up space for others, reducing the total area of re-configurable fabric required to implement the complete set of HWAs of a specific application. By allowing time-multiplexed use of limited silicon resources, smaller and less expensive devices can be utilised.

However, despite this DPR capability, the runtime management of available silicon area is not straightforward. Different HWAs implement different workloads (e.g., functions, or set of functions) with different memory and compute characteristics. Also, they may each require different silicon area, and have different average compute times. If a HWA is not currently available in the FPGA fabric, time must be expended to reconfigure the device. Memory transfer overheads, thermal considerations and problem size all influence the relative performance of the accelerator. Whether or not a specific task should be accelerated at a given time, or if execution should fall back to software thus depends on these aspects, on the particular task order and on current silicon usage, i.e. FPGA area.

1.2 Problem

As established, compute capabilities are no longer scaling with the physical node changes nor are these happening at the rate the industry enjoyed for the past decades. Heterogeneous architectures provide a way for better performance scaling and FPGA's characteristics place these devices at the forefront of these emerging architectures. The main cost of this approach is overall greater system complexity. As a result, software complexity increases as the effort required to manage the system also grows.

Most tasks related to managing heterogeneity are currently hand tuned. Task scheduling, kernel generation and the hardware/software partitioning problem are prime examples. This translates

into increased development costs as time is siphoned from developers that could be spend on more important tasks.

1.3 Objective

This work intends to explore techniques aimed at simplifying the process of developing and managing accelerated applications.

We believe that through the use of source-to-source compilation techniques one can effortlessly adapt an application to a heterogeneous environment and then resort to ML methods that learn how to appropriately schedule accelerated tasks. Given that decision trees are known to be efficient in terms of resource usage on FPGAs, we outlined the exploration of this specific ML mechanism to implement the runtime resource management.

In summary, we propose to address the following questions:

1. **Can tasks be scheduled on a heterogeneous system through machine learning algorithms trained at runtime?**
2. **Can source-to-source techniques be effective at transparently scheduling programs for heterogeneous architectures?**

1.4 Document Structure

This chapter contained a contextualisation of the theme presented in this dissertation. Chapter 2 elaborates on the topics presented, giving the reader the information required for understanding the rest of the document. Chapter 3 looks into past attempts by industry and academia at solving the problem we described. Chapter 4 details how we propose to solve that problem, building on top of previous knowledge, followed by chapters 5 and 6 detailing the implementation. Chapter 7 documents the experiments conducted and we conclude with chapter 8 offering an overarching view of this dissertation and the problems it addresses.

Chapter 2

Background

This chapter is a brief look into some of the concepts relevant to understanding this dissertation and the implementation choices made within it.

2.1 Field Programmable Gate Arrays

As the name suggests, [FPGAs](#) are a type of semiconductor device constituted by a array of logic blocks. These can be Lookup Tables ([LUTs](#)), Block RAMs ([BRAMs](#)), Digital Signal Processors ([DSPs](#)) and Flip-Flops ([FFs](#)). They are individually configured to make up individual logic operations that, when connected through a configurable interconnect, make up a comprehensive circuit. This circuit is then connected to Input/Output ([I/O](#)) blocks to ingest data and output results. To configure all these components, on startup, a bitstream is is loaded from the memory banks and applied onto the fabric.

Table 2.1: Full-adder logic table.

Inputs			Outputs	
<i>A</i>	<i>B</i>	<i>C_i</i>	<i>Sum</i>	<i>Carry</i>
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

An example of how a logic block emulates complex gate arrangements can be seen in the way [LUTs](#) work. Compared to a traditional digital circuit, the task of a group of logic gates between two registers can be assumed by a [LUT](#). For all possible inputs to that logic function, the [LUT](#) stores the outcomes and outputs them on demand (Table [2.1](#)). This simple mechanism allows a

LUT to be configured to assume the task of many different logic gates and condense them onto a single logic block.

The ability to configure a device at this level and interconnect the output of any block to another is what sets **FPGAs** apart. Even though the performance of an algorithm implemented on a **FPGA** will always be inferior to a direct implementation in **ASIC** form it still allows for improvements due to the use of special-purpose circuitry. This is the trade-off made with **FPGAs**.

That is why a common application for the use of **FPGAs** is **ASIC** prototyping as it allows a team of engineers to iterate on their logic design, test and validate it before incurring in the expense of doing a production run. Other applications centre around low volume circuit designs. Although **ASICs** are less costly per unit, the initial investment into design tools, expert teams, validation and non-recurring manufacturing costs make the cost proposal of **FPGAs** stand out for low production runs. Aerospace, automotive, medical or consumer electronics companies commonly use the reconfigurable nature of **FPGAs** to adapt the programmable fabric to their needs and implement custom circuitry at lower cost.

2.1.1 Benefits of special-purpose circuitry

Implementing algorithms using special-purpose circuitry allow us to make use of a few techniques aimed at enhancing the performance of that algorithm that would not be possible if it were running on a general purpose compute device (**CPU**). One example is parallelism. If two sequential task of an algorithm do not depend on each other results, they can be run in parallel to save time (Figure 2.1).

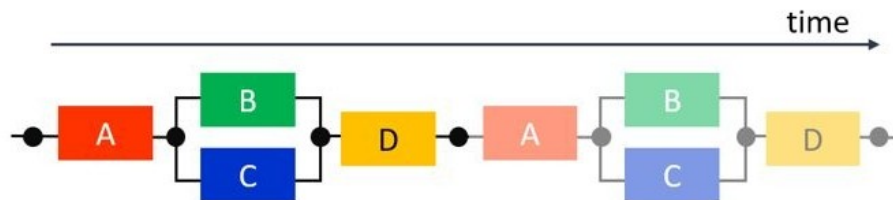


Figure 2.1: Task Parallelism within a Run. Source: Vitis Unified Software Development Platform 2021.1 Documentation.

Another improvement is task pipelining. If a run only needs each task at a single point of execution, the tasks can be pipelined. That is, as soon as each of them complete the processing of the current sample and pass the result on to the next task, they can already receive the next sample without waiting for the whole run to complete (Figure 2.2). This results in a reduction in the Initiation Interval (**II**) of an algorithm i.e. the time between sample ingests.

These enhancement strategies can then be applied simultaneously to maximise throughput as seen in Figures 2.3 and 2.4.

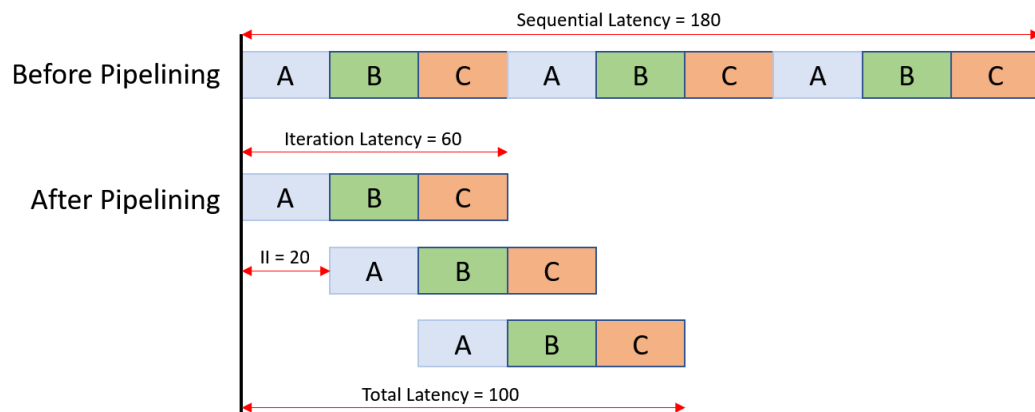


Figure 2.2: Task Pipelining. Source: Vitis Unified Software Development Platform 2021.1 Documentation.

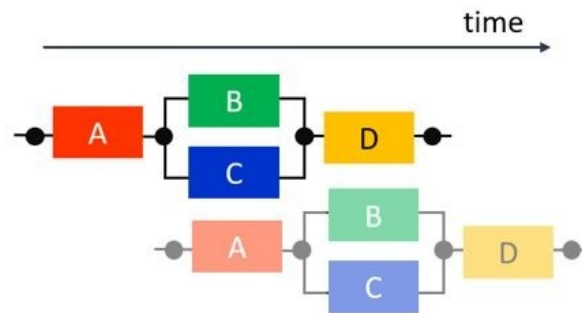


Figure 2.3: Task Parallelism with Pipelining. Source: Vitis Unified Software Development Platform 2021.1 Documentation.

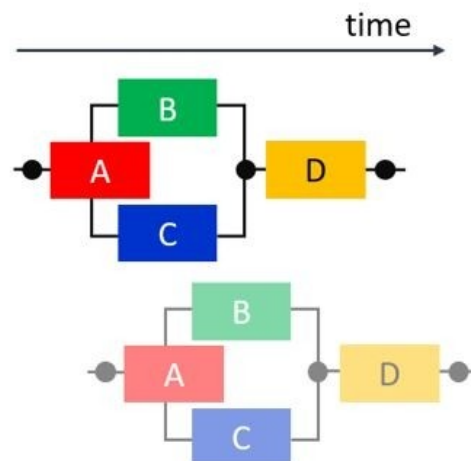


Figure 2.4: Task Parallelism and Pipelining within a Run, Pipelining of Runs, and Pipelining within a Task. Source: Vitis Unified Software Development Platform 2021.1 Documentation.

2.2 High-Level Synthesis

Traditionally, circuit specifications are done at the [RTL](#) level with [HDLs](#) such as Verilog or VHSIC Hardware Description Language ([VHDL](#)). This allows for fine-grained control of how the logic components of a circuit are arranged, how they function and how they are interconnected to perform a complex task. However working at the [RTL](#) level requires expert knowledge of how hardware architectures work and how the individual components will interact. While for many applications this working level is desirable, for others a greater flexibility and lower barrier of entry is required.

The use of C/C++ allows for development of algorithms at a higher level of abstraction. Engineers can implement the desired behaviour in C, C++ or OpenCL and validate the algorithms at that level without having to worry about hardware implementation details. Testing and simulation can be done using common software development tools leading to faster design iterations. [HLS](#) tools will then take this high level code and synthesise it to [RTL](#) level (Figure 2.5).

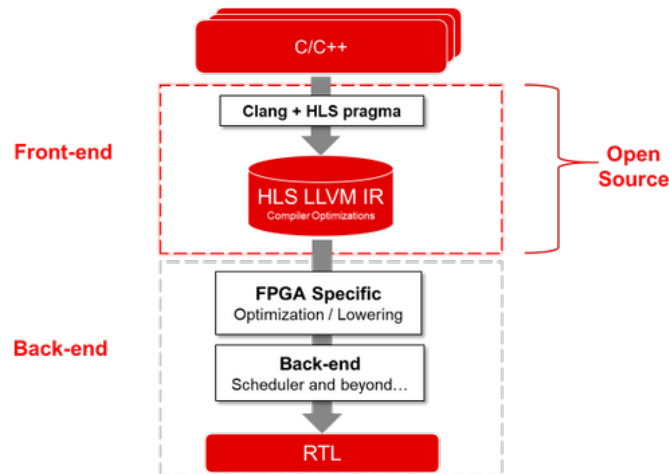


Figure 2.5: C/C++ to RTL synthesis flow in *Xilinx Vitis HLS*. It consists of 2 main components – 1. Front-end: This component parses the code expressed in C/C++ or OpenCL, applies front-end and middle-end transformations using the Clang/LLVM tool chain. 2. Back-end: This phase takes an LLVM IR input and performs FPGA-specific lowering and scheduling until the final step, RTL generation. Source: Xilinx Blog.

To guide the synthesis process, tools can interpret pragma directives inserted into the C/C++ code that guide optimisations steps. Examples include pipelining (Figure 2.2), array partitioning, loop unrolling, loop merging and function inlining. Developing in C/C++ allows for the rapid creation of multiple designs using different directives to conduct design space exploration and find an optimum solution.

Additionally [RTL](#) specifications are usually tied to a specific hardware architecture as the basic building blocks are not necessarily the same across device families or not available in the same quantity. [HLS](#) can be used to easily develop a design across devices and architectures and tailor optimised versions by again exploring the design space with different pragma directives for the different target devices.

HLS synthesis is divided into three main stages: scheduling, hardware resource binding and control logic extraction.

- Scheduling is used to determine when each operation will occur based on the target clock frequency, data dependency information extracted from the code, available resource, how long each logic block takes to complete an operation (varies depending on the target device) and on the user-defined pragma directives.
- Hardware Resource Binding allocates and maps each scheduled operation to the respective **RTL** implementations and thus device hardware resources.
- Control Logic Extraction creates a Finite State Machines (**FSM**) tailored to control the flow of data in the device as well as implementing the input and output ports.

Chapter 3

Related Work

3.1 Reconfigurable Hardware Resource Management Frameworks

In this section several academic efforts to develop hardware accelerator invocation and resource management are presented. These are used by applications to determine the best kernel implementation to utilise in order to obtain the fastest execution times.

3.1.1 SkePU

In the work of U. Daasteger et al. [11] the SkePU library is presented as being capable of generating code for accelerating a task when executed on a Graphics Processing Unit (GPU) using three different methods. Choosing the best implementation then becomes a problem. Recognising the non-trivial task that is performance portability between devices, the authors propose a method for automatic selection between the implementation variants.

The simplest approach would be to profile all possible variants of a function call context (number of arguments, their value and sizes) to determine the variant that provides the shortest execution time for each possibility. This, however, is unfeasible.

The empirical offline auto-tuning method the authors propose for implementation selection is based on a decision tree where the heuristic convexity assumption is applied. That is, in a 1-dimensional space, if two points A and B prefer the same implementation then all the points in the [A, B] range must also prefer it. This assumption can be expanded to a N-dimensional subspace where if a method is preferred for all vertices of that subspace then all points inside that subspace must perform better when using that same method. This algorithm is used to explore the parameter subspaces in a recursive manner.

Before every call to a function with several available implementations, the decision tree is evaluated in order to determine the most appropriate for the context of the call. In the case of SkePU, this can mean a execution purely on the CPU (C++) or on a GPU (OpenMP, CUDA or OpenCL).

Caution is given in the case of sequential kernel executions with tight data dependencies, as the algorithm makes the choice that provides the best performance for each individual kernel. This

combination, however, might not provide the fastest global execution as expensive data transfers may be required between kernel executions. To mitigate this issue, kernels are manually grouped and the framework executes the whole group on the device chosen for the execution of the first kernel.

The authors evaluated this approach by using nine benchmark kernels and demonstrated an effectiveness greater than 90% in predicting the best implementation from an exploration of just 0.5% of the possible training space. The authors point out that the method requires no modification of application-code when porting to new non-identical systems however offline retraining of the decision tree model is necessary using all new profiling information. Furthermore, being a library, SkePU requires users to be familiar with its concepts and programming model to be able to take advantage of its advertised capabilities. A program has to be designed with SkePU in mind from its inception.

3.1.2 Offload Annotations

In the work of G. Yuan et al. [12] a Python runtime is presented to assist in managing function offloading to **HwAs**. **GPU**s were used as an example. Specifically, the work focuses on popular Python libraries used in the field of data science.

Effectively, *annotators* write code for select library functions that indicates proper targets (accelerated substitutes) for each function. When the runtime decides that a function call is to be offloaded this target is called instead of the original **CPU** function. The offloading decision is made by the runtime based on cost estimates of data transfer and compute costs for either device. These estimators need to be globally defined by the user and can make use of the values and types of the arguments passed onto the library functions in their criteria. In their paper, the authors demonstrate the behaviour using simple linear estimators. In case they are not defined the runtime naively suggests using the accelerator if a target function has been provided otherwise the function is executed on the **CPU**.

Data management is also a major part of the runtime. As some datasets are able to fit in system memory but not in the memory banks of accelerators, the runtime uses user defined methods of dividing the input data to split this data into chunks manageable by the **HwA**. After executing the algorithms over all data chunks, data is transferred back to system memory and merged.

Tests were conducted on system with a **GPU** installed using it to accelerate Pandas, Sklearn and NumPy functions by means of the CuPy and PyTorch target functions. The results provided by the authors show that up to a 1200× speedup is possible with the median speedup being 6.3×.

With the possibility of estimators to be defined by the end-user, in theory, sophisticated algorithms can be used to guarantee portability of the code between every **CPU+HwA** meaning that if there are any hardware changes between systems no new profiling and tuning must be made.

One downside of this approach is that its usefulness is limited to situations where code reuse is common such as popular libraries. For every new kernel, a new annotation must be made. This is a feasible endeavour if it is done incrementally and over time. For generic programs written from scratch, Offload Annotations (**OAs**) are in essence the same as the current manual process. The

only advantages provided come from the data splitting and transfer management handled by the runtime.

3.1.3 Runtime Decisions

The work of G. Vaz et al. [13] describes a method based on profiling at runtime the performance characteristics of the **HwAs** and **CPU** when executing a task.

Making use of LLVM's ScalarEvolution (**SCEV**) analysis, information about loop trip counts is obtained. **SCEV** automatically determines if a trip count can be obtained statically and known at compile time or if it is dependant on the value of a variable at runtime in which case the decision on where to execute the kernel is deferred to when that value is available. This means that for deferred decisions, code is injected just before the function call to evaluate the loop trip count and determine if the computation should be offloaded.

Decisions are made based on user-defined thresholds. These need to be inferred by the user from profiling information and via manual tuning. The framework allows thresholds to be defined generally or be loop specific.

The authors show a speedup of up to 3× compared to a naive "**CPU** only" and up to 10× when compared to a "**HwA** only" approach on image processing workloads.

Two major limitations are apparent: 1) The dependency on the LLVM compiler framework [14]. This restricts development to programming languages supported by LLVM frontends and preventing use of compilers based on the GNU is Not Unix (**GNU**) ecosystem. 2) By doing the analysis steps and behaviour injection at the Intermediate Representation (**IR**) level a user is restricted to perform the compilation steps using the authors tools. If the user workflow includes using a closed-source LLVM-based compiler provided by the hardware vendor, this method is not applicable.

One final component to stress is the focus on loop trip counts. There is room for future research directions exploring diverse methods for inferring acceleration potential from the **IR** or other representations of source code.

3.1.4 QBC

In W. Ogilvie et al. [15] the Query by Committee (**QBC**) technique is proposed as a more efficient way of heuristic construction intended to aid in the creation of accurate **ML** models while requiring less training data.

The authors propose a method for choosing training data using interim **ML** models. This process starts by the creation of a set of 'seed' profiling data. These inputs are randomly selected. After offline profiling, 12 models are trained using different algorithms. These represent the 'committee'.

New, randomly generated points, are then evaluated by the 'committee' to derive new points of interest. The interest of each new point is inverse to the agreement of the 'committee'. If the models disagree on which implementation of a function should be used when a certain input

context is present then that context should be profiled and used for training. A new 'committee' is generated using the initial 'seed' data and the new 'candidate' points. This process is repeated until some completion criterion are met. The final set of points are used to train a final ML model for production use.

The authors show that using the proposed approach and avoiding profiling inputs that provide little to no information for a ML model, training overhead can be significantly reduced. An accuracy of over 85% was achieved on all examples while providing, on average, a 3x speedup of training due to the smaller training set used.

3.1.5 HIS

The Hybrid static-dynamic Implementation Selection (HIS) engine described in the work of D. del Rio Astorga et al. [16] takes advantage of annotations placed on C++ code to guide its behaviour. A user of this framework describes the available hardware for every function it wishes to accelerate. For each available HwA a variant must then be implemented to execute that function in the respective HwA. Information can be added to make sure that, for certain problem sizes, a specific HwA is used. For ranges where the user does not specify which one should be used, the framework collects profiling information and then uses it to add a *if-else-based* decision tree to the source code specifying the ranges where an implementation is preferred over the others.

3.1.6 Digest

In this subsection a short digest of the related work is presented. Table 3.1 collects the main points of the discussed works.

Table 3.1: Digest of decision engine characteristics

Section	3.1.1	3.1.2	3.1.3	3.1.4	3.1.5	5
Name	SkePU	OA	Runtime Decisions	QBC	HIS	<i>Tribble</i>
Type	decision tree	heuristic	threshold	heuristic	decision tree	Any
Retraining	Yes	-	-	Yes	-	-
Recompiling	Yes	-	Yes	Yes	Yes	No
Manual Code Changes	No	Yes	Yes	Yes	Yes	No
Compiler Lock-in	No	Python	LLVM	-	No	No

In the presented works, all decision engines require some sort of user interaction in order to port the applications to a different system. Due to the use of offline profiling and/or training, 3.1.1 and 3.1.4 require a period of exploration. This process could be automated on a per-device level as a bootstrapping period, but it would always require a waiting period between programming

and production use and depending on the possible exploration space can become impractical for embedded use. Alternatively, the end-user could train the models on a test system and then port the application and models to identical systems.

The works in 3.1.2, 3.1.3 and 3.1.5 all require code changes to be ported to different platforms. This would require source-code access and extensive manual profiling to determine the preferred methods of kernel execution.

3.2 Decision Trees

In the field of Decision Trees (DTs) there are many algorithms to choose from. A particular divide can be found in the type of output provided by DTs. Two great groups of algorithms can be observed: Classification and Regression. Classification algorithms try to learn the relation between the input values and provided labels. Any prediction places the input sample inside a particular class (e.g. scorching, hot, cold, freezing). Regression algorithms are more suited to learn relations to quantities (e.g. 50 through -10°C).

Another metric of classifying algorithms is by how they require the dataset to be presented. Again, two major categories emerge: Batch learners and Incremental learners. Batch learners require that the entire dataset is present in memory. The dataset is then split into a training portion and a testing portion. The algorithm constructs a model using the training dataset and then the fitness of that model is validated using the test dataset. A good training/test split is necessary to ensure that the model has not been overfitted. As one can imagine, memory requirements grow with the dataset size which makes many of these algorithms not suited for devices with restricted memory resources. Furthermore, if more data is added to the dataset a new model must be constructed from scratch. ID3 [17], and derivatives such as C4.5 and C5.0 are examples of DT batch learners.

Inversely, incremental learning algorithms as ID5 [18], ID5R [19] and ITI [20] allow for on-going learning from streaming data but store the dataset samples within the tree.

Hoeffding Trees [21] are incremental learning trees, which are more suitable for embedded scenarios because they have the following advantages: 1) They asymptotically guarantee the same classification as traditional batch learners. 2) They store information about the distribution of samples statistically rather than the samples themselves, which drastically reduces memory requirements, especially for large datasets.

A Hoeffding tree performs learning and inference by relying on a property of the Hoeffding bound that guarantees that best splitting point is chosen. If a gain function G , is to be maximised, then given $G(X)$ and $G(Y)$ - X and Y being the attributes that generate the highest and second highest values of G - if $G(X) - G(Y) > \epsilon$ then the Hoeffding bound guarantees that with probability $1 - \delta$ X is the best attribute to split on. The Hoeffding bound is computed as shown in Equation 3.1.

$$\varepsilon = \sqrt{\frac{R^2 \ln(\frac{1}{\delta})}{2N}} \quad (3.1)$$

3.2.1 VFDT

Very Fast Decision Tree (VFDT) is the original Hoeffding Tree implementation [21]. The statistical method used to store sample data is a binning system. These bins count the frequency of every value for analysed samples of the dataset. For categorical data, bins could either be assigned at compile time for every discrete value as the data range is a known parameter (it is required to calculate the Hoeffding Bound) or the number of bins can be capped and assigned to a value on a first-come-first-served basis as samples are seen by a node [22]. Once all the bins become assigned new values are discarded. For numerical data, the range can be divided equally among N bins. The value of N is a delicate balance. Too small and each bin covers an inversely large numerical range, being unable to distinguish small variations. Too large and the memory requirements may become great.

3.2.2 Gaussian Method

A method used mitigate the problems of the original bin system is to store information about sample values in the form of a Gaussian Distribution [23]. As the only values required to characterise the distribution of data for every attribute-class pair are the mean and variance, the data storage requirements are reduced heavily. This reduction is traded for an increase in computation requirements as one is no longer just incrementing bin values as samples come in.

3.2.3 Quantile Estimation

Zhe Lin et al. [24] makes use of asymmetric signum functions [25] for quantile estimation. This method uses each incoming sample to update a set of quantiles characterising the distribution of values for every attribute-class pair. The authors describe three main advantages of this method: 1) It does not require sample storage and the memory footprint is kept smaller than binning-based methods. 2) As only comparisons and subtractions are needed, there are less computational requirements than compared to the Gaussian Distribution method. 3) The algorithm is amenable for parallelisation and pipelining.

The authors proceed to describe an FPGA implementation written in Verilog for this algorithm. Experimental evaluation of the architecture granted a up to 1581× speedup over a server-grade CPU.

Chapter 4

General Approach

The intention of this work is to architecture a compilation flow capable of easing the entry of developers into the heterogeneous architectures space. To automate the process of taking a non-accelerated program and transforming it into an accelerated one with minimal developer intervention and where the end result is portable between different devices and architectures.

4.1 Scope

For the purpose of maintaining focus throughout the development of this work, restricting the scope of research is necessary.

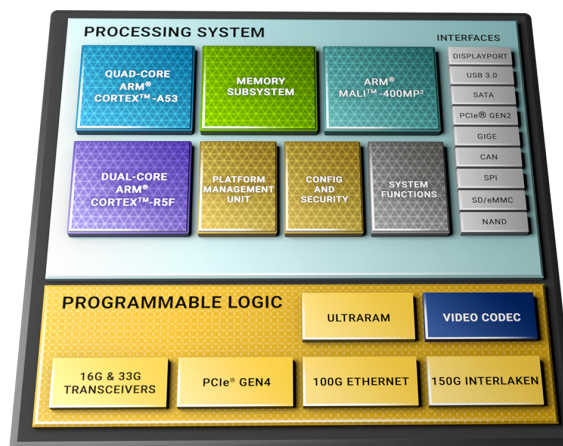


Figure 4.1: Diagram of internal architecture of Xilinx UltraScale and UltraScale+ family of integrated circuit devices. A processing system portion implemented as hard logic interfaces with the re-configurable area portion where accelerators are hosted. Source: Xilinx product pages.

1. We will only be working with Xilinx Multi Processor System on a Chip (**MPSoC**) products, that include **CPU** and **FPGA** portions, as shown in Figure 4.1. These systems are running a Linux-based Operating System (**OS**) (particularly Petalinux) with support for the OpenCL model of communicating and commanding the **FPGA** in the System on a Chip (**SoC**).

2. We do not intend to explore and expand on how to optimise code for HLS or conduct pragma space exploration.
3. We assume optimised version of the kernels can be generated either via expert intervention or by means of other tools or academic works.
4. We use the concept of arithmetic operation counts (an expansion on the notion of loop tripcounts) as an indicator of how long a task will take to compute. We do not intend to use it as a measure of absolute performance but as a means of comparing how they impact execution times on CPUs and FPGAs.

4.2 Proposed Architecture

The problem we chose to tackle can be divided into two main aspects: 1) A tool capable of transforming regular C/C++ programs and inject code to enable them to make use of accelerators. 2) A resource scheduler to choose between kernel implementations to optimise the overall application performance. The way they interact is described in Figure 4.2.

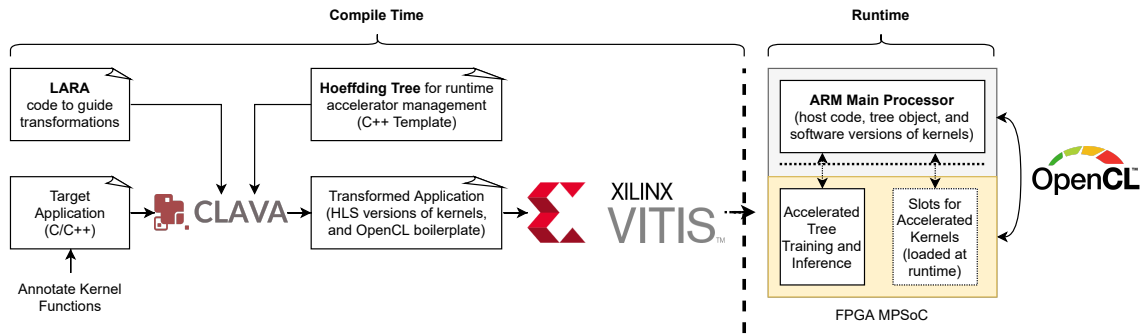


Figure 4.2: Diagram describing the proposed workflow.

4.2.1 Code Injection

In order to make the tool platform agnostic and portable, the best choice for where to make any modifications is the source code itself by means of the Abstract Syntax Tree (AST) allowing the program to be reconstructed into source code form to then be compiled by any toolchain as if it were written by the developer. A completely transparent process.

Clava [26] is a tool capable of being used as a base for this code injection as it automates the process of converting the source code into AST form, operate over the AST using a javascript-based Domain Specific Language (DSL) and simple Application Programming Interfaces (APIs) and finally convert the modified AST back to source code.

Any kernel implementation can also be used generate HwAs by means of HLS. Xilinx provides HLS support through its Vitis product [27] - the same software used to program the host platforms (CPU) of its embedded products.

4.2.2 Scheduling

State-of-the-art frameworks use loop tripcount estimations as input for their schedulers. We argue that this does not paint a clear picture as the content of each loop may influence runtime greatly. We intend to use arithmetic operation counts as the inputs to our scheduler (Algorithm 1). Specifically an array of 16 different counts spanning several bitwidths and operation types.

To guarantee portability between devices with the minimal developer intervention, offline profiling is not an option. Bootstrapping periods at a device's first usage is also something to be avoided, as the period would grow with the size of the exploration space and number of available HwAs. Due to these restrictions, ML is a compelling choice. Decision trees were chosen to the detriment of other ML algorithms as they are simpler to implement, amenable for FPGA acceleration and provide a way of easily extracting knowledge from. This last point is very important as to be able to guide future research into this field.

The intention is to use a C/C++ HLS implementation of a Hoeffding Tree with quantile estimation to learn, at runtime, what kernel implementation is preferable for executing a specific task.

Algorithm 1: Runtime Decision Engine

```

Initialisation;
Infer execution platform based on function call parameters.
Let  $\sigma$  be the confidence in the choice made by the tree model.
if  $\sigma > \text{confidence threshold}$  then
  | Execute task on preferred platform.
else
  | Start timer.
  | Start execution on CPU.
  | Use DPR to load the HwA (if needed).
  | Copy input data to HwA.
  | Start execution on HwA.
  | Wait for CPU or HwA to finish.
  | Halt execution of slower platform.
  | If applicable, copy result over from HwA.
  | Stop timer.
  if Train with sample then
    | Report winner to the tree model (training).
  end
end

```

Chapter 5

Tribble Source-to-Source Framework

As allured to earlier, the way in which we set out to create a tool capable of transparent code injection was through modification of the source code itself by means of the [AST](#). We dubbed this tool: *Tribble*. *Tribble* is a collection of scripts for the Clava source-to-source compiler. Clava provided an easy way to interface with the LLVM [AST](#) and modify any program without becoming tied to the LLVM infrastructure.

On a high level, *Tribble* achieves the following:

1. Automatically locate all functions tagged as kernels.
2. Transform any non-void kernel function into a void function.
3. Analyse kernel computational load.
4. Modifies the kernel function to allow for an external scheduler to decide which implementation to execute for every kernel function call.
5. Configures all OpenCL arguments and buffers required for [HwA](#) operation and inserts the respective OpenCL kernel-specific boilerplate invocation code without user intervention.
6. Injects code to profile the kernel executions and provide feedback to the external scheduler.
7. Replace the program's *main()* function with a version from predefined template while maintaining all functionality.

A quick note is warranted. The several code examples present on this chapter do not reflect parts of the same kernel. Each example is tailored to show as much information as possible about a specific feature to the detriment of providing a step-by-step guide to how a single kernel is morphed by each step taken by the *Tribble* scripts.

5.1 Folder Structure

In order to be able to use *Tribble*, a basic knowledge of how the tool is structured is required. Knowing where files are located is important for a user to be able to understand how *Tribble* functions and how to utilise its products further down the toolchain.

```

1  |-CxxIgnored
2  |  |-OCL_Helpers.cpp
3  |  |-OCL_Helpers.hpp
4  |-CxxSource
5  |  |-main.cpp
6  |  |-lib1.cpp
7  |  |-lib1.hpp
8  |-.clang-format
9  |-CxxTemplates
10 |  |-OCLH_main.cpp
11 |-woven_code
12 |  |-CxxIgnored
13 |  |  |-OCL_Helpers.hpp
14 |  |  |-OCL_Helpers.cpp
15 |  |  |_-HLS_graphs
16 |  |  |_-HLS_reports
17 |  |  |-CxxTemplates
18 |  |  |  |-OCLH_main.cpp
19 |  |  |-CxxSource
20 |  |  |  |-kernels.cpp
21 |  |  |  |-main.cpp
22 |  |  |  |-lib1.cpp
23 |  |  |  |-lib1.hpp
24 |_-Clava-Options
25 |_-Makefile

```

Listing 5.1: Relevant *Tribble* folder structure

Execution is simple as a Makefile is provided. `make run` and `make clean` are the available commands. `run` performs the code generation and `clean` deletes any artefact. As a final step to the code generation, the code is formatted using `clang-format`. For that, a `.clang_format` file is provided to allow for custom code styles.

- `CxxIgnored` - Contains source files that are not parsed by Clava. The files are just copied to the output folder without any modification. The purpose of this is to allow for *Tribble*-provided helper libraries to include C-style macros and enable their use in the code generation steps. The result is cleaner code, that is more readable, auditable and easier to debug.
- `CxxSource` - Main source code folder. Can be a symbolic link to the user's project.

- `CxxTemplates` - Internal *Tribble* C/C++ templates. Refer to section 5.9.
- `woven_code` - Output folder. All generated code is placed here. The folder structure of the input is maintained and reports from the HLSAnalysis step (section 5.10) are also present.

Any of names of any the folders used as inputs and outputs of code can be customised by altering the `Clava-Options` and `Makefile` files.

5.2 Kernel Pragmas

Tribble, as it stands, is not able to identify code amenable for acceleration. It falls outside of the scope of this work. Therefore, in order to have any knowledge of which function the developer wishes to turn into *HwA*, a couple of pragmas can be used to tag a function.

An additional pragma can be used as a guide for refining the OpenCL configuration of that kernel. The use of this second pragma is optional and its further explained in 5.7.

```

1 #pragma clava kernel
2 #pragma clava data kernel:[{ro:"auto"}, {wo:"auto"}]
3 void foo(int in[1000], int out[1000]) {
4     (...)
5 }
```

Listing 5.2: Pragma exemplification

5.3 Enforcing kernel void type

Injecting code for choosing between the kernel implementations can be done in two main ways:

1) Find all function calls for that kernel and wrap them in selection code. 2) Change the kernel function itself. With *Tribble* we chose the second as this minimises code changes and makes debugging easier. The original kernel code (Listing 5.3) is copied into another function and the original is modified to call the clone (Listing 5.4).

```

1 #pragma clava kernel
2 #pragma clava data kernel : [{ro : "auto"}, {ro : "auto"}]
3 int foo(int a, int b) { return a*b; }
```

Listing 5.3: Original kernel

```

1 int foo_KernelCode(int a, int b) { return a*b; }
2
3 int foo(int a, int b) {
```

```

4     (...)
5     int kernelReturn = foo_KernelCode(a, b);
6     (...)
7     return kernelReturn;
8 }

```

Listing 5.4: Code after initial transformation

Additionally, the OpenCL programming model forces every kernel to be of `void` type as all the memory management must be made manually. In order to support functions with `non-void` return types, an intermediate function is used to make the translation where the return variable is transformed into an argument. The script takes into account if the return type was already a pointer.

```

1 int foo_KernelCode(int a, int b) { return a*b; }
2
3 void foo_Kernel(int a, int b, int *kernelReturn) {
4     *kernelReturn = foo_KernelCode(a, b);
5 }
6
7 int foo(int a, int b) {
8     int kernelReturn;
9     (...)
10    foo_Kernel(a, b, &kernelReturn);
11    (...)
12    return kernelReturn;
13 }

```

Listing 5.5: Code after void type enforcement

As an alternative to this automated process, the user can manually define functions with these suffixes and they will be used by the script instead of creating them. This allows for custom modifications and expert use-cases not supported/provided by *Tribble*.

5.4 Estimating Kernel Compute Load

In order to determine if a function call should be accelerated, some information must be inferred about the computational requirements of that function call. To do so, a module called *StaticOpsCounter* is used to analyse the source code of the kernel and derive mathematical expressions for how many operations a kernel will compute.

The process starts by using Clava to look into the [AST](#) and finding every statement made in the scope of the function. There are two possible scenarios. The first being that the statement is not a loop. In this case, all the ops within this statement are counted and categorised according to their *type* (Integer Operation ([IOP](#)) or Floating Point Operation ([FLOP](#))), *bitwidth* and *kind* (sum,

multiplication, division). Additionally, if the statement is a function call, the module recursively calls itself. Conversely, if the statement is a loop the module is recursively called to analyse the loop's scope and an analysis of the loop headers is conducted.

```
1 for (uint i = 0; i < 10*58; i++) { (...) }
2 for (uint j = 0; j < N/2; j++) { (...) }
```

Listing 5.6: Loop Header Examples

A loop can be bounded by a constant or variable expression (Listing 5.6). In cases where the latter is true a check is performed to see if the variables in the expression are function arguments. If not, a search is made for the last time that variable was written to and what expression was used to do so. The variable is then substituted by the expression. This process is repeated until the loop expression is entirely reliant on function arguments, the algorithm gives up or it encounters a situation it is not capable of dealing with.

```
1 // Original
2 void foo (int N, int K) {
3     uint H = K/2;
4     uint J = N*H*5;
5     for (uint i = 0; i < pow(J, 2)/10; i++) {
6         (...)
7     }
8 }
9
10 // Loop count expression iterations
11 // pow(J, 2)/10
12 // pow((N*H*5), 2)/10
13 // pow((N*(K/2)*5), 2)/10
```

Listing 5.7: Loop Tripcount

Operation counts within the scope of a loop are then multiplied by the loop tripcount. The result is a series of expressions that easily calculate how many operations of each type, bitwidth and kind are needed to compute the requested function call. These expressions are then merged according to a map of similar expressions kinds.

```
1 static opEquivalenciesMap = {
2     // Custom op names for consolidation
3     sum: "sum",
4     free: "free",
5     // Names from the StaticOpsCounter whitelist
6     mul: "mul",
7     div: "div",
8     rem: "div",
```

```

9      add: "sum",
10     sub: "sum",
11     shl: "free",
12     shr: "free",
13     cmp: "free",
14     and: "free",
15     xor: "free",
16     or: "free",
17     l_and: "and",
18     l_or: "or",
19     mul_assign: "mul",
20     div_assign: "div",
21     rem_assign: "rem",
22     add_assign: "add",
23     sub_assign: "sub",
24     shl_assign: "shl",
25     shr_assign: "shr",
26     and_assign: "and",
27     xor_assign: "xor",
28     or_assign: "or",
29     post_inc: "add",
30     post_dec: "sub",
31     pre_inc: "add",
32     pre_dec: "sub",
33 }

```

Listing 5.8: Operation Kind Equivalences Map

After consolidation, the expressions are simplified and placed on a new function whose prototype is cloned from the original kernel. This guarantees that any function arguments used in the expressions are available to calculate them. An additional argument is added (`uint64_t opsCount[16]`) to be able to extract the calculated values.

```

1 void bar_KernelCount(int a, int b, uint64_t ops[]) {
2     ops[1] = b + 2 * pow(b, 2); // iops-32-mul
3 }

```

Listing 5.9: KernelCount Example

In this example, the kernel function `bar` was analysed, and an auxiliary function `bar_KernelCount` was created. By receiving the same argument list, the function can compute, at runtime, the expected compute load of the kernel as a function of calling argument values. This information is used by the scheduler to infer which device would deliver the lowest execution time.

5.5 Scheduling

As established, the selection of what implementation to use for a kernel call is carried out by a scheduler. This is a function call (that can be a [HwA](#) itself) inserted into the kernel that utilizes the operations counts calculated beforehand in `foo_KernelCount` as input for a decision.

```

1  extern void scheduler(uint64_t ops[], bool &executeOnCPU, bool &executeOnFPGA,
2                          bool &measurePerf);
3
4  // The function "foo" is processed by the source-to-source compilation stage, and
   is replaced with the following result. The function signature is preserved, and
   the original function is called based on whether the scheduler determines if
   the execution is faster on CPU or FPGA.
5  int foo(int a, int b) {
6      int kernelReturn;
7      bool executeOnCPU;
8      bool executeOnFPGA;
9      bool measurePerf;
10     uint64_t opsCount[16] = {0};
11     foo_KernelCount(a, b, opsCount);
12
13     // The scheduler may determine to execute the kernel on both targets if
       learning is required, or otherwise infers which target is best from past
       data and executes only on that platform.
14     scheduler(opsCount, executeOnCPU, executeOnFPGA, measurePerf);
15     if (executeOnCPU) {
16         (...)
17         foo_Kernel(a, b, &kernelReturn);
18         (...)
19     }
20     if (executeOnFPGA) {
21         (...)
22     }
23
24     return kernelReturn;
25 }
```

Listing 5.10: Scheduler Insertion

There are three boolean variables controlled by the scheduler. The first two, `executeOnCPU` and `executeOnFPGA`, control where the kernel should be executed (it is possible to select both). A third variable `measurePerf` signals whether or not the scheduler is confident in its decision and therefore requests profiling of the execution.

5.6 Profiling and Training

To provide feedback and train ML-based scheduling algorithms, information about the performance of a kernel's execution is needed. *Tribble* achieves this by using the C++ `chrono` library to measure the execution time of the kernel in either target.

```

1  extern void scheduler(uint64_t ops[], bool &executeOnCPU, bool &executeOnFPGA,
2                          bool &measurePerf);
3  extern void train(uint64_t ops[], bool CPUwonFPGAlost);
4
5  int foo(int a, int b) {
6      int kernelReturn;
7      bool measurePerf;
8      std::chrono::high_resolution_clock::duration clava_timing_duration_0;
9      std::chrono::high_resolution_clock::duration clava_timing_duration_1;
10
11     scheduler(opsCount, executeOnCPU, executeOnFPGA, measurePerf);
12     if (executeOnCPU) {
13         std::chrono::high_resolution_clock::time_point clava_timing_start_0;
14         std::chrono::high_resolution_clock::time_point clava_timing_end_0;
15         if (measurePerf) {
16             clava_timing_start_0 = std::chrono::high_resolution_clock::now();
17         }
18         foo_Kernel(a, b, &kernelReturn);
19         if (measurePerf) {
20             clava_timing_end_0 = std::chrono::high_resolution_clock::now();
21             clava_timing_duration_0 = clava_timing_end_0 - clava_timing_start_0;
22         }
23     }
24     if (executeOnFPGA) {
25         std::chrono::high_resolution_clock::time_point clava_timing_start_1;
26         std::chrono::high_resolution_clock::time_point clava_timing_end_1;
27         if (measurePerf) {
28             clava_timing_start_1 = std::chrono::high_resolution_clock::now();
29         }
30
31         // Execute on FPGA
32
33         if (measurePerf) {
34             clava_timing_end_1 = std::chrono::high_resolution_clock::now();
35             clava_timing_duration_1 = clava_timing_end_1 - clava_timing_start_1;
36         }
37     }
38     if (measurePerf)
39         train(opsCount, clava_timing_duration_0 < clava_timing_duration_1);
40
41     return kernelReturn;
42 }

```


Listing 5.11: Profiling Code

Each kernel execution is wrapped in code that calculates the duration of those executions and then relays that to the external `train` function in the form of a boolean result that can be used to improve future scheduling decisions, without the need for profiling, even if the particular combination of calling arguments has not yet been observed for the kernel.

5.7 OpenCL Kernel Arguments

One of the most complex code generation tasks is the management of the OpenCL API calls. As the programming model is heavily reliant on expert intervention, automating the interface between [CPU](#) and [FPGA](#) quickly grows in complexity. In particular, data transfers must be managed manually.

As mentioned in [Section 5.2](#), an optional `pragma` can be used to aid this process. Omitting the `pragma` outright, or partially, results in the script falling back to the default behaviour (`auto`).

```
1 #pragma clava data kernel : [{auto : "auto" }, {scalar : "auto" },
2                               {rw: "N*sizeof(int) "}]
```

Listing 5.12: Buffer pragma

The content of this `pragma` is a JavaScript Object Notation ([JSON](#)) object detailing how a kernel's argument is used and how much space does it occupy in memory. Each object (`{}`) in the array (`[]`) maps to an argument in order of declaration. Several options are available to indicate how the argument's memory will be used: read-only (`ro`), write-only (`wo`), read-write (`rw`), automatic detection (`auto`) and that the argument is a scalar and is not modified during the execution of the kernel (`scalar`). When no information is provided, `auto` is used. Currently `auto` maps to `rw` however future work includes analysis capabilities to infer if a variable is read or written to.

For each key used to indicate how the memory is used a value must `auto` be provided. The `auto` option is also available here. It results in an expression where the type of the argument is wrapped in a call to `sizeof()`. Examples can be seen in [code listing 5.13](#). Alternatively, users can provide a C/C++ expression as seen in [listing 5.12](#).

Buffers are only created for non-scalar arguments and have the `CL_MEM_USE_HOST_PTR` option enabled in order to reuse the memory objects passed into the kernel as arguments as opposed to allocating memory for this purpose.

```
1 cl::Buffer buffer_a(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
2                             sizeof(int[2]), a);
3 cl::Buffer buffer_b(context, CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY,
```

```

4         sizeof(float), &b);
5 cl::Buffer buffer_c(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_WRITE,
6         a[0]*sizeof(unsigned long long int), &c);

```

Listing 5.13: Buffer Examples

After all buffers are created, they need to be set as arguments to the [HwA](#). An example of this can be found in listing 5.14.

```

1 extern cl::Kernel krnl_foo;
2
3 krnl_foo.setArg(0, buffer_a);
4 krnl_foo.setArg(1, b);

```

Listing 5.14: Argument assignment

Finally, all that is left is to transfer all necessary buffers between devices, execute the kernel and transfer the results back.

```

1 #pragma clava kernel
2 #pragma clava data kernel : [{scalar: "auto"}, {ro, "auto"}, {rw, "auto"}]
3 int foo (int a, int B[100], int C[100]) {
4     (...)
5     queue.enqueueMigrateMemObjects({buffer_B, buffer_C}, 0 /* 0 means from host */)
6     ;
7     queue.enqueueTask(krnl_foo);
8     queue.finish(); // Wait for the kernel to finish executing before transferring
9     the results back
10    queue.enqueueMigrateMemObjects({buffer_C}, CL_MIGRATE_MEM_OBJECT_HOST);
11    queue.finish();
12    (...)
13 }

```

Listing 5.15: Data transfer and kernel execution

API calls for enqueueing tasks are non-blocking. Therefore, the C/C++ code proceeds execution without waiting for the data transfers to finish, nor the kernel execution. Because of that, it is necessary to wait for the command queue to become empty before transferring the results back from the [HwA](#).

The `krnl_*` variables are `cl::Kernel` objects created externally and imported as global variables to make them available inside any required function. This is also the case for the `cl::CommandQueue` and `cl::Context` objects needed at various locations.

This process of adapting an arbitrary annotated kernel to conform with the calling convention of our target execution device is thus fully automated by the source-to-source step of our flow. This is, on its own, a significant contribution to the usability of Xilinx [FPGA](#) devices. Although Vitis

HLS offloads the hardware design effort from the programmer (albeit with non-optimal results), there is still a hard requirement for potentially extensive code re-writing in order to conform with the OpenCL based abstractions between the processing system and the reconfigurable fabric, which this contribution of this dissertation circumvents. Additionally, this process conforms with current limitations and requirements imposed by the [HLS](#) compiler, as outlined following.

5.8 Vitis Incompatibilities

For compiling C/C++ [HLS](#) kernels, the Vitis software utilises the v++ compiler. Some peculiarities of this compiler introduce a few impediments to the source-code generation. One of the checks v++ performs is if the C/C++ [HLS](#) code conforms to the rules and restrictions of the Xilinx [HLS](#) implementation. One of them being that no calls to operating system are permitted (as a kernel would not have access to the [OS](#)).

Part of the process of generating [HwAs](#) using Vitis includes selecting which C/C++ functions we wish to accelerate. It happens that v++ does not apply the [HLS](#) restrictions only to these functions and code they depend on. Instead it applies them to the entirety of the input, meaning that if the code happens to import a library that does not comply with the [HLS](#) restrictions the compilation fails. This is the case if we import the OpenCL headers in order to be able to use the API.

A solution to this problem is to separate the `*_Kernel` and `*_KernelCode` functions from the rest of the codebase into a different file. This allows us to give the v++ compiler this new file as the input. The program is still compilable for the host platform (executing on the [CPU](#)) as *Tribble* creates prototypes for the `*_Kernel` and `*_KernelCode` functions in the header files of each respective library. The linker will then join the program appropriately.

5.9 `main()` Template

As alluded to before in section [5.7](#), several variables and functions related with the OpenCL programming model used in the files where kernels reside and are defined are created externally to those files. These originate from a file present in the `CxxTemplates` folder that includes a function named `main_template`. The purpose of this file is to serve as a template for any program being accelerated using *Tribble*. It contains the code that creates and configures all OpenCL objects, imports the `xclbin` file, finds the [FPGA](#) device and defines the `scheduler` and `train` functions.

```

1 #include "../CxxIgnored/OCL_Helpers.hpp"
2
3 cl::Context context;
4 cl::CommandQueue queue;
5 cl::Program program;
6

```

```

7 void scheduler(uint64_t ops[], bool &executeOnCPU, bool &executeOnFPGA,
8               bool &measurePerf) {
9     executeOnCPU = true;
10    executeOnFPGA = false;
11    measurePerf = false;
12 }
13
14 void train(uint ops[], bool CPUwonFPGAlost) { return; }
15
16 #pragma clava ocl_insert_globals
17 // cl::Kernel krnl_foo;
18
19 int main_template(int argc, char *argv[]) {
20
21     static const std::string platformName = "Xilinx";
22
23     OCLH::getConfig(argv[1], platformName, context, queue, program);
24
25 #pragma clava ocl_insert_kernel_initializations
26     // krnl_foo = OCLH::getKernel("foo_Kernel", program);
27 }

```

Listing 5.16: Example of configuration file

In Listing 5.16 a few new things stand out. The use of functions from the OCLH namespace and the presence of two pragmas. These pragmas are used as markers for where to insert code pertaining to the creation and configuration of the `cl::Kernel` objects. Below them are examples of the code that is created by *Tribble*. The OCLH namespace is defined in the `OCL_Helpers.hpp` file. This library is composed of a number of helper functions with the purpose of simplifying the process of configuring the OpenCL environment.

A simple solution to insert this configuration code at the beginning of the program execution without disrupting the main function is to replace the function altogether as the main. Thus, when *Tribble* is executed, main becomes `main_original`, `main_template` becomes main and calls `main_original` to execute the intended program behaviour.

```

1 // CxxSource/main.cpp
2 // main --> main_original
3 int main_original(int argc, char *argv[]) {
4     (...)
5 }
6
7 // CxxTemplates/main.cpp
8 extern int main_original(int argc, char *argv[]);
9
10 // main_template --> main
11 int main(int argc, char *argv[]) {

```

```

12     (...)
13     return main_original(argc, argv);
14 }

```

Listing 5.17: main() replacement mechanism

5.10 HLSAnalysis

As a final step, all `*_KernelCode` functions are submitted to the Clava script developed by Tiago Santos in his master's dissertation entitled 'HLSAnalysis' [28]. The script performs an analysis of the source code and places `HLS` pragmas intended to maximise the performance of that code when synthesised. For information regarding how that script achieves this, please refer to his work.

Summarily, the automatic HLS conversion introduces partitioning pragmas and pipelining and/or unrolling based on data dependencies.

5.11 Trials and Tribble-ations

Throughout the development of *Tribble*, limitations were noticed about the approaches used and code generated. This section discusses them, explains why they exist, and proposes possible future solutions for most of said limitations.

1. Xilinx OpenCL does not allow the use of `const` in the arguments of a kernel. This can be solved by removing the type qualifier in `*_Kernel` functions. As it is still present in the original function prototypes it will not affect the program behaviour. The detection and removal of the `const` qualifier can also be used to augment the `auto` setting described in section 5.7 as any argument with it is known to not be modified by the code and thus can be tagged as read-only (`ro`).
2. Xilinx OpenCL does not allow the use of the `static` storage class on a kernel. When a function is tagged as a kernel using the pragma described in section 5.2, *Tribble* should remove it.
3. Class methods are a particularly interesting problem. As the Xilinx OpenCL only allows communication with the kernel through its arguments, there is no way to reference an object. A possible solution would be to create a static method on the same class where a parameter is added. That parameter being a pointer to an object of that class. Let's call this Solution 1. The issue with Solution 1 is that, as described before, the `static` storage class cannot be used. A quick fix to this problem would be to extract the kernel from the class scope. Define it as a standard function. Let's call this Solution 2.

Solution 2 would compile and execute but has a slight flaw. It does not account for the possibility of two functions having the same name. If a method `foo` from classes `A` and

B is tagged as a kernel, two functions with the same name, but different arguments will be created outside the namespace of their respective classes. OpenCL does not allow two kernels to have the same name as it uses the kernel names to identify them. Solution 3 then is to cryptographically hash the prototype of the function and use it as its name or randomly generate a string. We believe this would prevent any naming collisions and allow for the use of class methods seamlessly in *Tribble*. Hashed or randomly generated values would then be converted to strings with 25 alphanumeric characters, as a prefix is needed to insure that the function name starts with a letter (C/C++ requires it).

```

1 // Original
2 int A::foo(int x);
3
4 int A::foo_KernelCode(int x);
5
6 // Solution 1
7 static int A::foo_Kernel(A* obj, int x) {
8     obj->foo_KernelCode(x);
9 }
10
11 // Solution 2
12 int foo_Kernel(A* obj, int x) {
13     obj->foo_KernelCode(x);
14 }
15
16 // Solution 3
17 int Kernel_1E6FpfZkDcbp7mTfk2dvZeFna(A* obj, int x) {
18     obj->foo_KernelCode(x);
19 }

```

Listing 5.18: Class method solutions

4. As hinted in the solution to the previous limitation, OpenCL restricts the length of the name of kernels to 32 characters. By applying the aforementioned solution to all kernels and not only class methods, this problem can be avoided.
5. As described in section 5.8, Vitis forces the entirety of the program inputted to v++ to follow the HLS rules and restrictions. This includes code from external libraries. However, kernels may depend on these external libraries to function. This way, header files present in the original files where the kernels were located are copied to the separate file described in that section. Users are then required to remove any incompatible includes that are unnecessary for the kernels to operate.
6. Currently the `auto` option described in section 5.7 defaults to `rw`. In cases where a kernel parameter is never written to and is not an array, `auto` should instead classify it as a scalar.

If it is an array, `ro` would be more appropriate. The same principle can be applied to parameters that are only written to.

7. Currently, OpenCL buffers use the pointer to original variable passed as an argument to the kernel (section 5.7). As this saves time by not requiring the data to be copied to a new location, it does not guarantee that it is 4096 bit aligned. This results in more time expended transferring data between the CPU and FPGA as additional memory copy operations are required. Two solutions are possible. Create 4096-aligned variables inside the kernel function and spend the time to copy the data over or locate where the variables are created in the first place and guarantee they are aligned. Instinctively, the first option seems less error-prone as `structs` and `unions` would rapidly complicate how the second solution would need to work. Either way, experimentation would be required to determine the best balance.

8. As it stands, if the scheduler requests that a task should be executed on both the CPU and FPGA and that they be profiled this happens sequentially. The end goal of *Tribble* is to have them running concurrently and when one of the platforms finishes the execution on the other is terminated early. This would minimise the overhead of learning.

Additionally, if the kernel stores results on the same input buffer, running on CPU and FPGA for training will result in an erroneous result as the data will suffer more modifications than expected. This can be solved by detecting if a parameter is not a scalar and is not `const`. That being the case, the data should be cloned before any computation occurs. This ties into one of the solutions proposed for the previous limitation.

9. In `*_KernelCount` functions, the types of the parameters can influence the result of the calculation. Overflows can occur, intermediate negative values saved to unsigned variables, etc. A possible solution would be to force the conversion of all parameters to floating point data types. The output would then be done in these types instead of large unsigned integer types.
10. Either manually or as part of the processing done by `HLSAnalysis`, array partitioning pragmas can be inserted into the code. Partitioning the function parameters will result in additional OpenCL kernel arguments (`krnl.setArg()`) proportional to how the variables are partitioned. In this cases, splitting the data and creating the additional transfer buffers can be done but it is not currently implemented.
11. `StaticOpsCounter` (described in section 5.4) currently does not support recursive functions. This is not a relevant limitation as *HLS* itself disallows the use of recursive functions but a limitation nonetheless.
12. `StaticOpsCounter` currently does not support loops of any type other than `for` loops. That means that any kernel containing a `while` or `do...while` loop will result in an exception being thrown.

13. StaticOpsCounter does not take into account the use of `break` or `continue` inside loops. If these statements are dependant on the value of data being worked on by the kernel then no action can be taken. However, if they depend only upon the values of the arguments passed to the kernel then the expression used to evaluate whether or not to execute the statement can be incorporated into the expressions used to estimate the loop tripcount.
14. StaticOpsCounter is currently only counting arithmetic operations and discarding any information about memory access.
15. The module referred in section 5.4 used for expression simplification is currently limited and does not support many mathematical constructs.

5.12 Toolchain

One of the goals laid out in chapter 4 was to establish a transparent process that slots in between the code development and industry tools. This is easily done in *Tribble*. In this section the example of Xilinx Vitis 2020.2 is shown.

5.12.1 Project Creation

Vitis provides a set of examples and templates for projects. A simple one that creates all the project folders required is the *Hello World (HLS C/C++ Kernel)* from the *SW acceleration templates*. This example is only available if the chosen hardware platform supports use of the Xilinx Runtime (XRT). Figure 5.1 illustrates one such platform. Special attention is warranted for the value in the 'Flow' column and the domains available. The sysroot and rootfs used in the project configuration will also impact which examples are made available.

After completing the configuration in the wizard, the folder structure in listing 5.19 is what is expected to be found in the Vitis workspace folder. The project name 'ExampleProject' has used for this demonstration.

```

1 // Other files and folders were omitted as they are irrelevant for this discussion.
2
3 |-ExampleProject // Project for code running on the host (CPU)
4   |-src
5   |   |-host.cpp
6 |-ExampleProject_kernels // Project for C/C++ HLS kernel code and HLS synthesis
7   |-src
8   |   |-vadd.cpp
9 |-ExampleProject_system
10 |-ExampleProject_system_hw_link // Kernel synthesis and bitstream generation

```

Listing 5.19: Example Vitis project folder structure

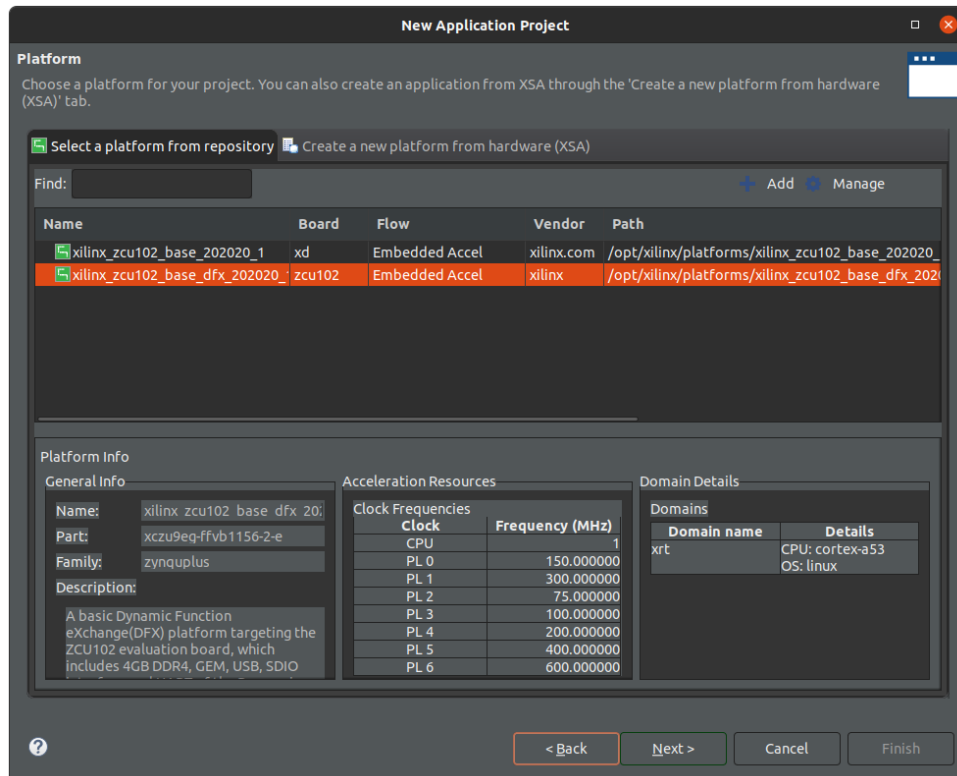


Figure 5.1: XRT support in Vitis platform

5.12.2 Tribble output integration

An easy way to pipe the output of *Tribble* to the Vitis projects is through symbolic links. Listing 5.20 exemplifies the two required links.

```

1 | -ExampleProject
2 | -src // Symbolic link to 'woven_code'
3 | -ExampleProject_kernels
4 | -src
5 | | -kernels.cpp // Symbolic link to 'woven_code/CxxSource/kernels.cpp'

```

Listing 5.20: Vitis projects required symbolic links

With the symbolic links in place, all that remains is to select the functions to be accelerated in Vitis. That is achieved in in both the 'ExampleProject_kernels.prj' and 'ExampleProject_system_hw_link.prj' files using the Graphical User Interface (GUI).

The projects can now be compiled.

Chapter 6

Hardware Resource Scheduler

As described in chapter 5, the *Tribble* framework prepares a program for the use of accelerated kernels. Tasks are executed on these [HwAs](#) when a scheduler considers it can provide a result faster than the [CPU](#).

In order to prevent the need for a lengthy exploration of the problem space to determine the execution times of every single kernel or a bootstrapping period to train an algorithm on a device's first use, we intend to make use of the Hoeffding Tree's properties to take a learn-as-you-go approach to the resource scheduler.

In this dissertation, we present a flexible C/C++ [HLS](#) implementation of a Hoeffding Tree variant tailored for use in FPGAs, originally proposed by Lin et al. [24]. Their work built atop an earlier variant in which the storage of the statistical data of the sampling distribution of the original Hoeffding Tree was replaced by a Gaussian approximation [23]. Lin et al. replace this approximation with quantile estimation using asymmetric signum functions [25]. The result is a larger memory footprint but a reduction in computational requirements, while achieving similar results. Since it is implemented in Verilog, the applicability of the implementation is limited to circuit synthesis, e.g. for FPGA.

By using [HLS](#), we expand upon previous work by providing a parametrizable implementation that is equally suitable for CPU and FPGA allowing for synergies between compute resources, easy customisability and support for a wide range of tree dimensionalities.

6.1 HLS Hoeffding Tree Implementation

We implemented the tree as a parametrizable C++ class. Specifically, the parameters include the maximum number of nodes in the tree, the number of attributes of each data sample, and the floating-point precision. The class contains a training and inference methods which are the target of HLS synthesis. At runtime, the C++ tree object can be manipulated in software, and passed as an argument to the training/inference method. This introduces the potential for instantiating several tree objects in memory, which can be processed by the same synthesised circuit, assuming

the same template parameters. Additionally, training and inference can be dynamically partitioned between software and hardware, although this exploration is out of the scope of this dissertation.

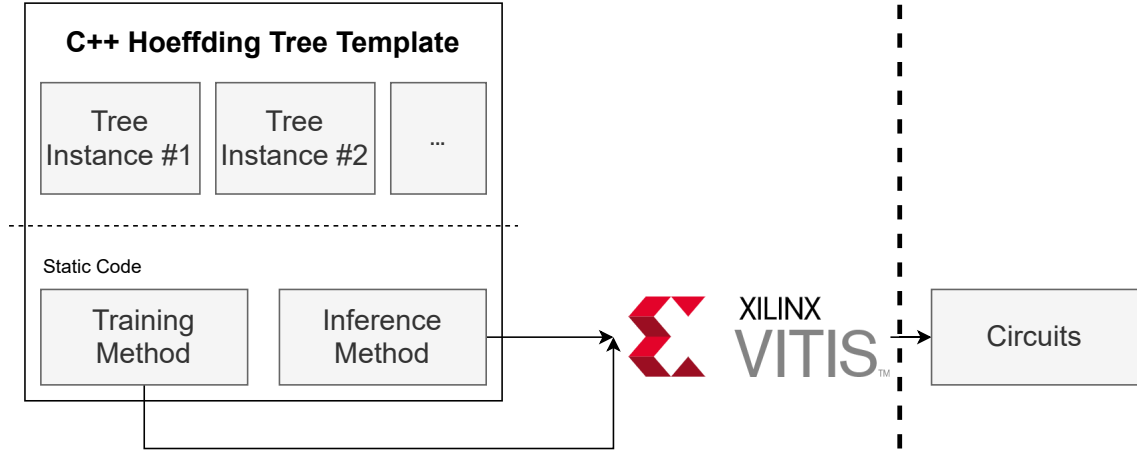


Figure 6.1: Hoeffding Tree kernel generation and workflow diagram.

As discussed, we are using a Xilinx Vitis HLS flow that enforces the use of OpenCL [APIs](#) for the invocation of the synthesised kernels. The implemented C/C++ kernel dubbed `krnl_Tree` receives 4 arguments: a `HoeffdingTree` object, an array of samples structures, an array of structures that store the tree inferences and the size of this array.

When using the OpenCL model, a large overhead would be introduced if the hardware tree kernel were to be invoked for one sample at a time, as the processing time of a very low volume of data is overwhelmed by a large transfer time. A practical application of the kernel design could be, for example, in the sensor domain, where the tree could continuously sample fused data from multiple sensors (i.e., multiple attributes) without host processor intervention and therefore [API](#) related overheads. Alternatively, to mitigate this overhead, the host may choose to postpone the classification of samples until it has accumulated a sufficiently large number such that the volume of data mitigates this overhead.

Inference on an incremental learning decision tree cannot be easily parallelised as the model changes and evolves with every training sample that arrives. This restricts the pipeline to dealing with one sample at a time, sequentially. The sample structure contains information about whether it should be used for training purposes or only for inference. Thus, as the kernel loops through the sample array, it executes either the `train` or `infer` method of the tree object accordingly. The results are placed in the output data structure.

The OpenCL [API](#) allows for fine-grained control of how these arguments are passed to the kernels. Each one is a separate buffer, and the transfer of these buffers to and from FPGA memory is done manually by the user. While touted as a downside previously in this work, this fine-grained control allows for one to transfer the tree object to FPGA memory and not retrieve it between executions, as the memory persists until the user changes it. With this mechanism, a model can be trained on the FPGA and then retrieved for analysis at a later stage.

Models can also be swapped between kernel executions, opening the door for decision tree ensembles. Trees can reuse the same kernel instance for a time-multiplexed ensemble or several copies of `krnl_Tree` can be instantiated in parallel and, by assigning the same sample buffers to all of them, multiple trees can process the same data without memory duplication.

Our code is structured in layers, allowing for modularity in the tree construction. The following C++ classes are used.

6.1.1 NodeData

This templated class is responsible for storing all the data and methods regarding training in a node. In this case, it harbours all methods for quantile estimation, how to calculate the Gini impurity of the node and find the optimal split candidates. Template parameters of this class allow for compile-time customisation of the type used for storing the quantile values and make non-integer calculations. Defaults to `float` type. Changing this type affects the precision of all the tree's calculations. Other customisations include the number of tree attributes (D) and output classes (K).

6.1.2 Node

The `Node` class stores all information regarding a node in the tree: whether it is split, what are its children, the split value, split attribute and the corresponding Data object. The template parameters for this class allow for changes to the capacity of the tree in terms of maximum amount of nodes (as it impacts the type used to store node indexes) and to the Data object class (defaults to `NodeData`).

If one decides that the Gaussian approximation method is preferable in their case, a reimplementation of `NodeData` is all that is necessary.

6.1.3 BinaryTree and HoeffdingTree

`BinaryTree` is a base class for binary tree operations. It stores an array of node objects (whose type is defined in the class template), contains methods for managing those nodes (splitting a node and defining children) and sort a sample through the tree.

The `HoeffdingTree` class extends `BinaryTree` to include methods on how to calculate the Hoeffding bound (Equation 3.1) and the higher-level training algorithm that is agnostic to how the data is physically stored and managed.

6.1.4 TypeChooser and TypeChooserMath

`TypeChooser` is a helper library that supplies macros for automatic integer type selection. The macros take as inputs the minimum and maximum values expected to be stored in a variable (just the maximum for unsigned types) and returns the smallest type that can fit those values. This is only compatible with C++ as the macros make use of `std::conditional` directives. The main

value of these macros is that they are evaluated at compile time and therefor can be used with [HLS](#) and in templates to select the most appropriate data types.

TypeChooser is also compatible with Xilinx Arbitrary Precision ([AP](#)) types. A check for the definition of the `USE_XILINX_AP_TYPES` macro is used to select whether [AP](#) types will be returned by the signed and unsigned macros or if they are restricted to native C/C++ types. Bitwidths of up to 64 bits are supported.

The use of [AP](#) types introduces compatibility issues with mathematical functions of the `std` namespace. Xilinx provides its own `hls` namespace for mathematical functions compatible with its [AP](#) and native C/C++ types but a host of problems arise with its use as hardware implementations of mathematical functions for native types are available but not any software implementations. TypeChooserMath is a companion library to TypeChooser that provides a namespace (`tc_m`) that is host to a number of wrapper functions aimed at resolving these conflicts. Usage of [AP](#) types is still discouraged for beginners as the `hls` hardware implementations are based on different principles than those used by the standard library (`std`) and may not provide the same results.

6.2 Tree Visualisation

A major factor for the use of Decision Trees as the scheduler in this dissertation is the ability to extract usable knowledge from them - provided that their scale does not defy the limits of human understanding. To achieve this goal a method for visualising the tree models is a necessary aid.

The availability of mature, robust and feature-complete libraries in Python has made it the *de facto* language for [ML](#) tasks. Tensorflow, Keras, Pytorch, SciKit Learn, Matplotlib and ONNX are a few examples of the tools that power the Python [ML](#) ecosystem. As companions to this environment, a suite of visualisation tools are available that interface with the formats used by these libraries. To capitalise on this rich tool ecosystem, we implemented the `JsonExporter` C++ class capable of exporting a Hoeffding Tree model to the format used by SciKit Learn. Specifically, the tree model is exported to [JSON](#) in a format compatible with the `sklearn_json` Python library [29] which in turn is capable of interpreting it and constructing a `sklearn.tree.DecisionTreeClassifier` model.

6.3 Limitations

The presented implementation of the Hoeffding Tree algorithm is host to a few limitations. A rather impacting one is the fact that the type used for attribute storage is also the one used for intermediate, fractional, calculations. This forces the attribute storage type to be floating point and thus potentially larger than what would be required to hold all the possible values for those attributes.

Chapter 7

Experimental Evaluation

7.1 Experimental Setup

Our evaluation targeted the Xilinx ZCU102 Evaluation Board [3] featuring the Zynq UltraScale+ XCZU9EG-2FFVB1156 MPSoC. Programs were compiled using the Xilinx Vitis IDE v2020.2.0. All Vitis projects were configured to use the *xilinx_zcu102_base_dfx_202020_1* prepackaged platform by Xilinx resulting in the use of the Xilinx Zynq MP First Stage Boot Loader Release 2020.2 and a prepackaged PetaLinux distribution using the Linux kernel version 5.4.0-xilinx-v2020.1.

The clustering data sets used for testing our Hoeffding Tree implementation were synthetically generated with different numbers of points, clusters, and dimensions. Attributes are randomly correlated. With these datasets we did not intend to evaluate classification accuracy, but instead to present the [FPGA](#) resource requirements of the synthesised training and inference methods for several template parameters and evaluate the execution time versus the on-chip ARM Cortex-A53 (1.2GHz) [CPU](#).

Tests involving the *Tribble* framework follow the setup and compilation flow detailed in section [5.12](#) and use kernels from the Polybench Suite.

7.2 Validation Threats

We identify a few situations that may pose a threat to the validation of the research described in this documents, introduce biases into the experiments and thus reduce the meaning of their results.

1. There is no publicly available implementation (in C/C++ or otherwise) of the Hoeffding Tree utilising quantile estimation by means of asymmetric signum functions. Thus we have no ground truth to validate our results against and cannot guarantee that our implementation of the tree is free of algorithmic or coding errors.

2. Validation of the usability of the *Tribble* framework was made using small examples or benchmarks from the Polybench Suite [30]. Tests have not been conducted with comprehensive, real-world examples of programs actually intended to be deployed in embedded devices on a production environment.

7.3 Evaluation

In this section we present the following experiments:

1. An evaluation of the throughput of our implementation of the Hoeffding Tree, for **CPU** and **FPGA**, using synthetic data clusters.
2. An evaluation of the size of the Hoeffding Tree object in **CPU** memory, and of the **FPGA** resource usage of the respective train/inference methods, as a function of the tree's K and D parameters.
3. We compare our Hoeffding Tree implementation with a state-of-the-art implementation (Lin et al. [24]), for two reference datasets.
4. A test of the full-flow compilation ability of *Tribble* with an example from the Polybench benchmark set.
5. We demonstrate that the output of *Tribble* is in fact valid and benchmark the generated code.
6. An evaluation of the Hoeffding Tree's ability to learn from arithmetic operation counts using hypothetical kernels.
7. We propose and evaluate a scheduling algorithm based on a Hoeffding Tree.

7.3.1 Hoeffding Tree Throughput

These results were obtained by feeding the tree with datasets of K clusters in a D dimensional spaces, constituted of N points (Figure 7.1). These were generated using the work of Paulino [31]. For these experimental runs, we will have the entire dataset transferred in a single operation to the **FPGA**'s memory.

Looking at the first four rows of Table 7.1 (D=3) it can be observed that for a 3-dimensional dataset, regardless of the bundle size, the ARM **CPU** in the ZCU102 **SoC** significantly outperforms the **FPGA** implementation in both the training and inference tasks. Also, the performance gap between both implementations grows with the number of samples processed. This indicates that the kernel is slower, per iteration, than the pure software solution.

Regarding the last four rows of Table 7.1 (D=100), the picture is not quite as clear. When a dataset with a larger number of dimensions is used, the ARM **CPU** still outperforms the **FPGA** kernel in training. However, it does so with a lower margin and one that does not appear to grow with the added number of samples. On the inference task with this larger dataset, the **FPGA** outperforms the ARM processor by 8.4×.

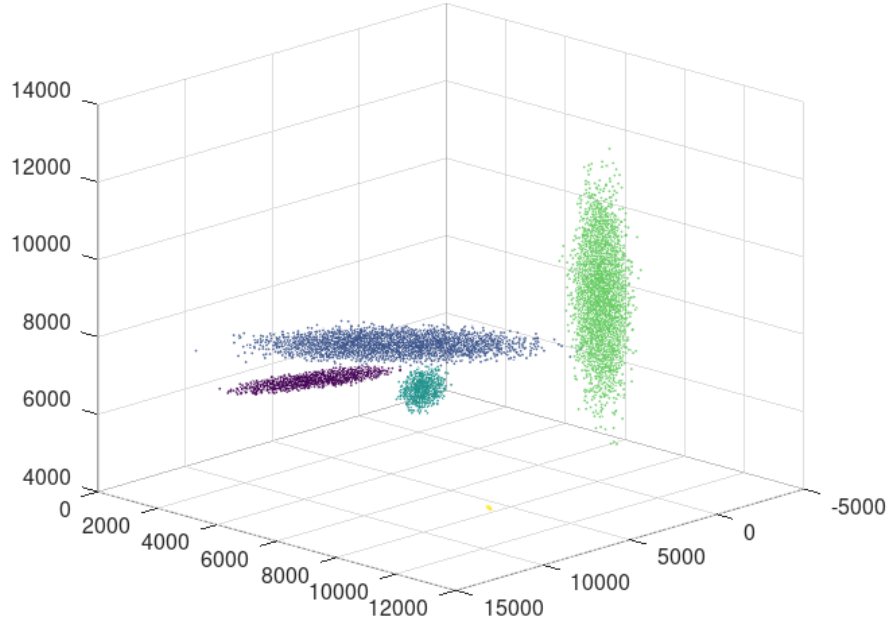


Figure 7.1: Visualisation of a 3 dimensional ($D=3$), 5 cluster dataset ($K=5$), with 40k points ($N=40,000$). The number of points in each cluster is random.

7.3.2 Hoeffding Tree Resource Utilisation

Table 7.2 presents various configurations of the kernel, tailored for datasets of different dimensions (D), with different number of classes (K), number of samples (N) and max number of nodes (N_d). The purpose is to determine the effect of these parameters on [FPGA](#) resource utilisation. As expected, parameter N has no effect on resource utilisation as samples cannot be processed in parallel.

When it comes to the other parameters, they all result in an increase in resource usage. This is due to the highly sequential nature of the generated kernel, which also explains why the performance of this kernel on training tasks is poor compared to the [CPU](#). This overall advantage is less surprising when considered in the context of an 11-fold [CPU](#) advantage in clock speed. Current [HLS](#) tools cannot automatically parallelize sequential code. Without hardware design expertise in order to optimise the design, the implementation will be far from optimal. In our implementation, we still believe that further parallelization can be achieved even within a single tree, through inner loop unrolling or memory partitioning.

One interesting result is that of the kernel's operating frequency. It remains unchanged for all configurations. Looking deeper into the cause of this phenomenon, we find that the bottleneck is the sorting of a sample down from the root node to the appropriate leaf node. A process with queue depth of one. This sequential operation also prevents the kernel from being pipelined.

Figure 7.3 presents the size of the tree object for the ranges of parameters used. The maximum value measured was of 64.2 MiB. Each variable has a linear effect on object size growth.

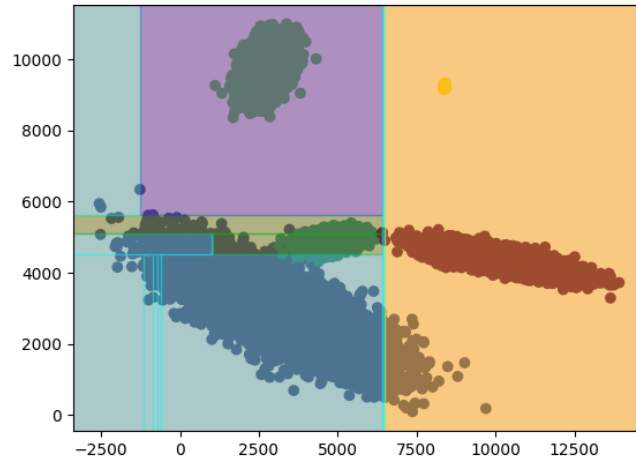


Figure 7.2: Two dimensional projection over the Z-axis of how the dataset from figure 7.1 was classified by a tree.

Table 7.1: Training and inference times for four synthetic clustering datasets, for the ARM CPU (1.2Ghz) and the FPGA (103MHz).

K	D	N	Task	ARM CPU	FPGA	Speedup
3	40k		Training	207 ms	1,990 ms	0.10×
			Inference	151 ms	462 ms	0.33×
	500k		Training	2,983 ms	30,933 ms	0.10×
			Inference	2,260 ms	11,442 ms	0.20×
5	40k		Training	6,028 ms	51,648 ms	0.12×
			Inference	3,924 ms	469 ms	8.37×
	100		Training	75,763 ms	651,775 ms	0.12×
			Inference	49,495 ms	11,494 ms	4.31×

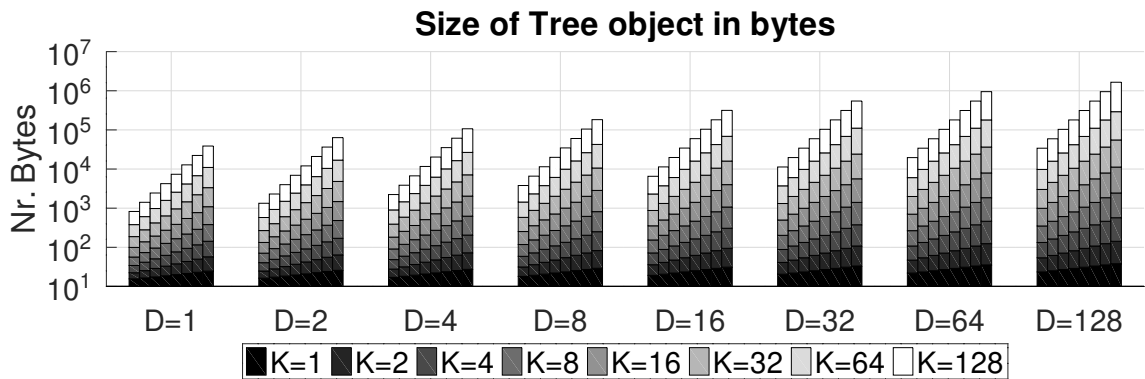


Figure 7.3: Evolution of the size of Tree objects in bytes by changing Nd, D and K. Each bar in every grouping, depicts a tree with a max number of nodes from 2^0 to 2^7 . Datatype is `float`.

Table 7.2: N, D, K and Nd effects on FPGA Resource Utilisation

Nodes	100	100	100	1000	100	100	100	1000
K	5	5	10	5	5	5	10	5
D	3	100	3	3	3	100	3	3
N	40k	40k	40k	40k	500k	500k	500k	500k
LUT	23304 (8.6%)	20567 (7.6%)	23776 (8.8%)	24351 (9.0%)	23304 (8.6%)	20567 (7.6%)	23776 (8.8%)	24351 (9.0%)
LUTRAM	1395 (1.0%)	1179 (0.8%)	1399 (1.0%)	1397 (1.0%)	1395 (1.0%)	1179 (0.8%)	1399 (1.0%)	1397 (1.0%)
FF	35682 (6.6%)	29775 (5.5%)	36374 (6.7%)	36336 (6.7%)	35682 (6.6%)	29775 (5.5%)	36374 (6.7%)	36336 (6.7%)
BRAM	12 (1.3%)	9.5 (1.0%)	12 (1.3%)	12 (1.3%)	12 (1.3%)	9.5 (1.0%)	12 (1.3%)	12 (1.3%)
DSP	23 (0.9%)	25 (1.0%)	25 (1.0%)	25 (1.0%)	23 (0.9%)	25 (1.0%)	25 (1.0%)	25 (1.0%)
BUFG	13 (3.2%)	13 (3.2%)	13 (3.2%)	13 (3.2%)	13 (3.2%)	13 (3.2%)	13 (3.2%)	13 (3.2%)
MMCM	1 (25.0%)	1 (25.0%)	1 (25.0%)	1 (25.0%)	1 (25.0%)	1 (25.0%)	1 (25.0%)	1 (25.0%)
Freq. (MHz)	103.6	103.6	103.6	103.6	103.6	103.6	103.6	103.6

7.3.3 Hoeffding Tree Kernel Performance

To evaluate the accuracy of our C/C++ HLS implementation, we obtained results for two of the UCI datasets used by Lin et al. [24].

Table 7.3 presents the benchmarks the following tree parameters were used (same as in Lin et al. [24]): $\delta = 0.001$, $\lambda = 0.01$, $\tau = 0.05$, $n_{min} = 200$, $n_{pt} = 10$, $n_{quantiles} = 16$, $Nd = 2047$, with one being of special relevance: Nd (maximum number of nodes).

When compared to the author’s implementation, our kernel presents a significant slowdown. With the increased number of nodes, the sequential tree traversal portion of the algorithm increases in length. Our HLS implementation achieves comparable accuracy for *Bank*, although the performance for *Covertypes* is inferior. Lin et al. [24] reports 89.30% and 72.51%, respectively. We believe a difference in calculation precision between the CPU and FPGA caused the degradation in *Covertypes*, despite the use of 32-bit floating point data types for both devices.

Table 7.3: Training time and Accuracy (Acc.) for Covertypes and Bank datasets, for the ARM CPU (1.2Ghz) and the FPGA (103MHz)

	Lin et al. [24]	ARM CPU		FPGA		
		Acc.	Time	Acc.	Time	Speedup
Bank	89.30%	88.25%	202 ms	88.25%	8,525 ms	0.02×
Covertypes	72.51%	72.21%	9,712 ms	63.71%	374,600 ms	0.03×

7.3.4 Tribble Code Transformations

We tested the flow described in section 5.12. The example used was the *adi* benchmark from the Polybench suite. In appendix B a full example of the use of *Tribble* is present with integral file contents to show how the steps documented in chapter 5 translate to a complete application. For this run, the expression simplification module was not used as per limitation 9 documented in section 5.11.

A few alterations to the source code were necessary to make the final program compile (for the CPU and FPGA) and behave correctly.

1. The types of the parameters `tsteps` and `n` in the `kernel_adi_KernelCount` function needed to be modified from their original type (`int`) to `uint64_t`. This is again due to limitation 9.
2. As the C++ compiler mangles all function names during compilation, when Vitis searches the AST for the original name to start the synthesis steps, it is unable to find it. To prevent this issue from happening, the `kernel_adi_Kernel` function definition in the `woven_code/CxxSource/kernels.cpp` file needed to be wrapped by a `extern "C"` block.
3. Neither original `adi.h` and `adi.cpp` files include a prototype for the `kernel_adi` function. Due to this peculiarity, when that original function is cloned during the execution of

Clava, no prototype for `kernel_adi_Kernel` is inserted in the output files. The code shown in listing 7.3.4 had to be manually inserted into the `woven_code/CxxSource/adi.cpp` file.

4. The HLSAnalysis processing step (section 5.10) introduced a number of HLS pragmas in the `kernel_adi_KerneCode` function. Among these were `array_partition` pragmas applied to the function arguments. In its current form, *Tribble* is not able to deal with the effects of these pragmas when partitioning the function arguments as per limitation 10 described in section 5.11.

```

1 extern "C" {
2 extern void kernel_adi_Kernel(int tsteps, int n, float u[2000][2000],
3                               float v[2000][2000], float p[2000][2000],
4                               float q[2000][2000]);
5 }

```

7.3.5 Execution of Generated Code

The code resulting from the previous compilation flow experiment was executed in the ZCU102 board from where the following experimental data was obtained. A variation of that compilation flow was also created where the datatype used by the kernel is `double` as opposed to `float`.

Figures 7.4 and 7.5 plot the profiling data collected by *Tribble* with the code described in section 5.6 pertaining to the performance of the *adi* kernel. These demonstrate a clear performance advantage by the CPU where the FPGA implementation consistently takes 40× more time to complete the same task.

Figures 7.6 and 7.7 paint a similar picture. These refer to the `float` version of the kernel. In this configuration the performance gap gets reduced but the CPU is still the clear winner and the trendline of figure 7.7 that for the maximum possible value of N (2000) that situation will not change.

To boost FPGA performance, light changes were made to the kernel code (listing C.1). This narrowed the performance gap even further to a point where the FPGA still produced 6× worse execution times than the CPU.

As shown, HLS is no substitute for expert manipulation of code and optimisation for FPGA environments. We demonstrate that slight modifications to the codebase, can lead to major improvements in a kernel's performance on a FPGA device.

7.3.6 Hoeffding Tree - Learning from Operation Counts

In order to evaluate a resource scheduling algorithm, a necessary premise is that the algorithm is forced to choose. That is, the same choice must not be the optimal for all regions of the possible exploration space. If, as seen in the previous experiment, a CPU always outperforms an FPGA

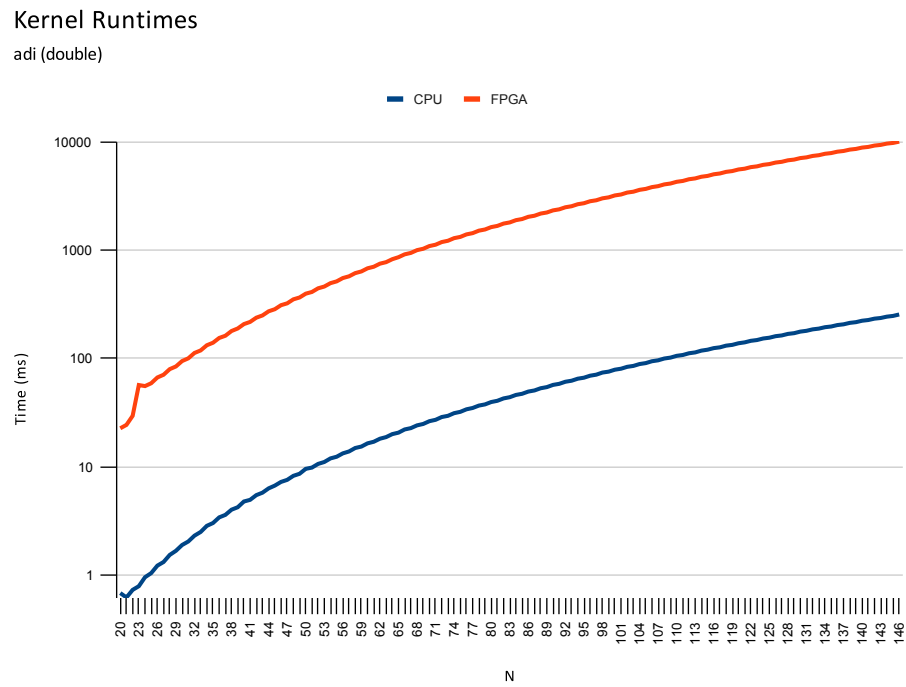


Figure 7.4: Evaluation of *adi* kernel runtimes on the the CPU and FPGA portions of the ZCU102 board using 'double' as the kernel datatype. $tsteps = N/2$. $N \in [20; 146]$

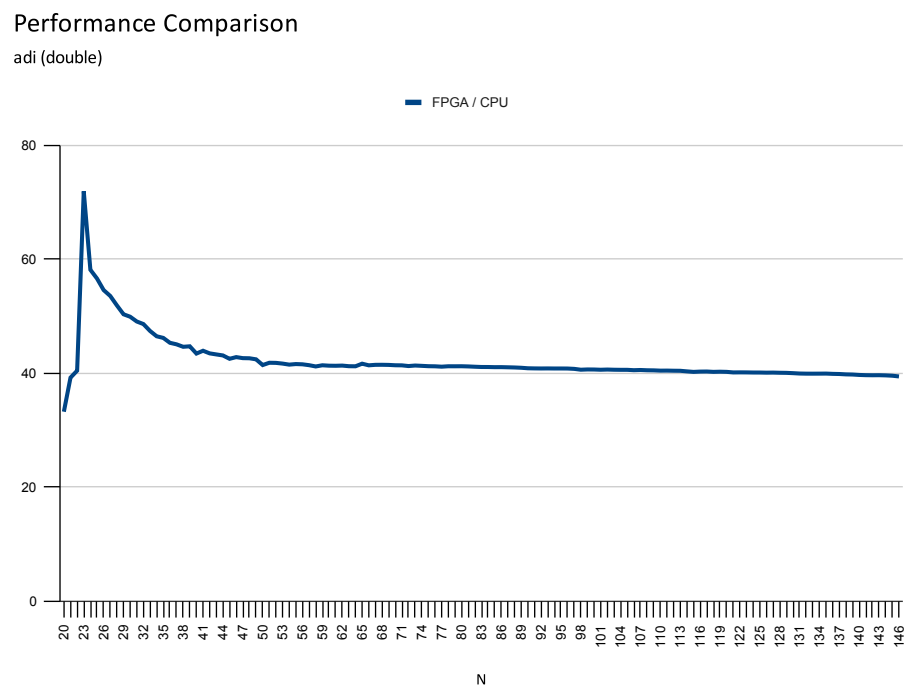


Figure 7.5: Comparison between the FPGA and CPU execution times for the *adi* kernel using 'double' as the datatype. $tsteps = N/2$. $N \in [20; 146]$

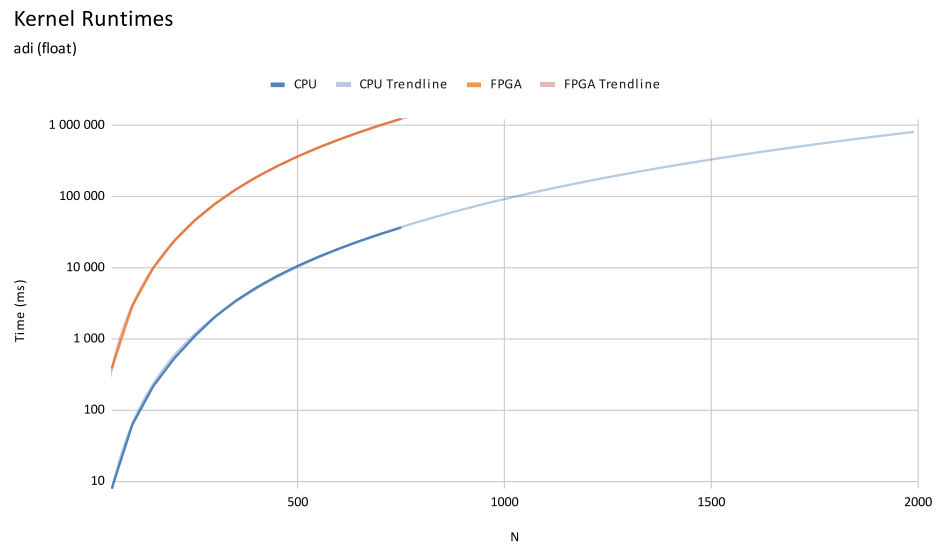


Figure 7.6: Evaluation of *adi* kernel runtimes on the the CPU and FPGA portions of the ZCU102 board using 'float' as the kernel datatype. $tsteps = N/2$. $N \in [50; 750]$

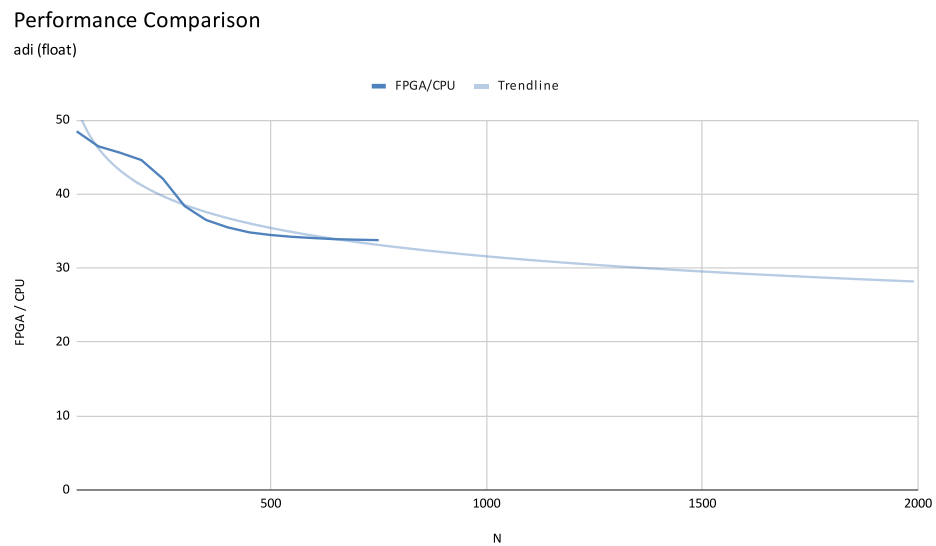


Figure 7.7: Comparison between the FPGA and CPU execution times for the *adi* kernel using 'float' as the datatype. $tsteps = N/2$. $N \in [50; 750]$

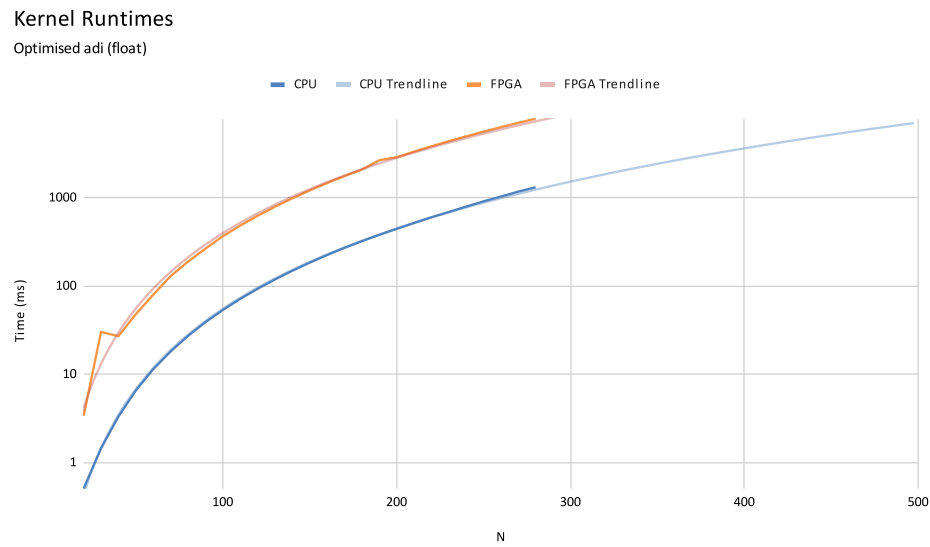


Figure 7.8: Evaluation of *adi* kernel runtimes on the the CPU and FPGA portions of the ZCU102 board using 'float' as the kernel datatype. $tsteps = N/2$. $N \in [20; 280]$. Kernel has had light manual optimisations made to the code.

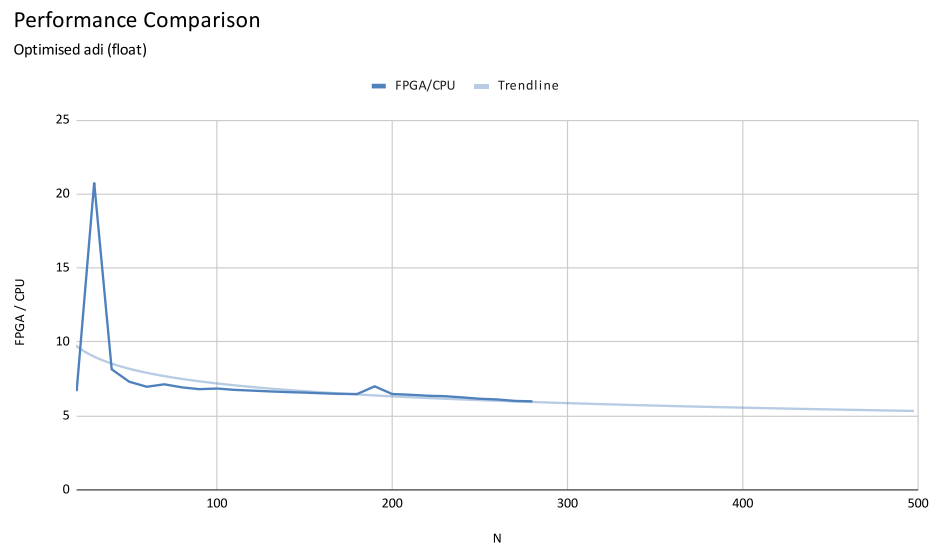


Figure 7.9: Comparison between the FPGA and CPU execution times for the *adi* kernel using 'float' as the datatype. $tsteps = N/2$. $N \in [20; 280]$. Kernel has had light manual optimisations made to the code.

implementation then there is nothing for the scheduler to decide and much less a [ML](#) algorithm to learn. For an experiment to be valid at least one region of the exploration space must exist where the [FPGA](#) outperforms the [CPU](#).

A possibility would be to simulate a device less powerful than the ZCU102. This could be done by reducing the operating frequency of the [CPU](#). This would not an unreasonable thing to do to guarantee that our poorly optimised kernels beat the [CPU](#) as most embedded devices do not sport such a high-end part. Reducing the frequency to half would roughly double the time the [CPU](#) takes to complete a task.

Another possibility would be to, either manually or through source-to-source techniques, optimise the kernels for [HLS](#) synthesis. This tuning would involve considerable effort and falls outside of the scope of this dissertation.

In the end we chose not to follow either of these approaches. Instead, we will be using hypothetical kernel optimisations. Let us use the *adi* kernel again as an example. We can assume that, with sufficient optimisation, we are able to synthesise a version of *adi* that for half of the problem space has a lower runtime on the [CPU](#) and that on the other half performs better on the [FPGA](#).

For the *adi* kernel, only two parameters control the problem size. These are n and *tsteps* (Listing 7.1). The polybench suite already proposes a relation between the two variables ($tsteps = n/2$) in the default dataset sizes. We can use this relation to make a uni-dimensional exploration space.

```

1 void kernel_adi(int tsteps,
2               int n,
3               double u[2000][2000],
4               double v[2000][2000],
5               double p[2000][2000],
6               double q[2000][2000]);

```

Listing 7.1: *adi* kernel prototype.

With $n \in [20; 2000]$ where a kernel call performs better can be determined by $n > ((2000 - 20)/2)$ with 0 (`false`) representing the [CPU](#) and 1 (`true`) the [FPGA](#). These principles can be applied to other kernels (*2mm*, *atax*, *bicg* and *deriche*) to create a more comprehensive dataset. This method allows for the simulation of the behaviour to be guaranteed to be correlated to the operation counts. The full code used to create the dataset can be consulted in Appendix D along with the relations between input variables and operation count expressions for all the kernels.

Note that using this method it is possible for two equal feature sets (same operation counts vector) to produce different results. This adds a dimension of variability to the dataset that reflects the internal characteristics of each kernel. e.g. kernel A can be more parallel than kernel B and thus for the same feature set kernel A should execute on the [FPGA](#) while kernel B should be executed on the [CPU](#). For that reason, we chose to train six models in total. One for each of the five individual kernels and one overarching model with all the data points. Note that there is no

Table 7.4: Dataset characteristics for hypothetical optimised kernels.

Model Name	Nr. Samples	Better on FPGA
All Kernels	12761	52.26%
<i>adi</i>	1981	50.98%
<i>2mm</i>	2185	50.71%
<i>atax</i>	2169	51.45%
<i>bicg</i>	2169	51.45%
<i>deriche</i>	4257	51.49%

feature in the feature set that identifies a kernel. The characteristics of each dataset can be seen in Table 7.4

Table 7.5 presents the accuracy results for all kernels. The feature set values were normalised to fit into a $[0; 1]$ range. This was done by dividing all values by 40,000,000,000. However for actual deployment one of the following functions could be used to bound unbounded variable such as the operation counts:

$$\frac{2}{\pi} * \arctan \frac{x}{A} \quad (7.1)$$

$$\tanh \frac{x}{B} \quad (7.2)$$

$$\frac{2}{1 + e^{(-x/C)}} - 1 \quad (7.3)$$

A, B or C should be adjusted to the appropriate value.

Interleaved testing-then-training was used for this measurement as it is a common method used for evaluating incremental learning algorithms. Before every run, the order of the samples on the dataset is randomised to minimise the effect the order has on the model. Additionally, each experiment has 3 runs. The final accuracy considered is the average of all the runs.

In Table 7.5 the accuracy value for the third run of the *2mm* dataset is highlighted. The discrepancy between the other runs of this kernel is due to the fact that on the first two, the tree only managed to split very late in the execution (Figure 7.10a) as opposed to the highlighted execution where only 600 samples were required to make a split (Figure 7.10b). To curb this problem, the experiment was repeated reducing the number of samples required to attempt a split on a leaf node (Tables 7.6 and 7.7). We can observe an increase in the accuracy for *adi* and *2mm* and a massive jump for *deriche*.

With this example we have demonstrated that, when the exploration space has regions where the CPU is preferable and where FPGA is preferable, it is possible to use arithmetic operation counts as the means to construct a tree model capable of selecting the platform providing the fastest execution (kernels *adi*, *2mm* and *deriche* in Tables 7.6 and 7.7). For kernels *atax* and *bicg* the tree was unable to split using only the available samples. The model targeting all five kernels also was unable of producing satisfying results. Future exploration is warranted with a larger dataset or with an added feature that identifies each kernel.

Table 7.5: Hoeffding Tree accuracy results for hypothetical optimised kernels. Leaf nodes do split trials on every 200 samples collected.

Model Name	Average Accuracy	Individual Runs		
All kernels	65.99%	64.75%	68.04%	65.18%
<i>adi</i>	50.40%	50.93%	49.92%	50.33%
<i>2mm</i>	65.17%	58.86%	54.19%	82.47%
<i>atax</i>	51.07%	51.31%	50.76%	51.13%
<i>bicg</i>	50.21%	48.92%	51.13%	50.58%
<i>deriche</i>	57.21%	55.27%	60.21%	56.14%

Table 7.6: Hoeffding Tree accuracy results for hypothetical optimised kernels. Leaf nodes do split trials on every 50 samples collected.

Model Name	Average Accuracy	Individual Runs		
All kernels	64.12%	63.90%	67.26%	61.21%
<i>adi</i>	70.54%	69.71%	72.14%	69.76%
<i>2mm</i>	77.47%	82.47%	74.64%	75.29%
<i>atax</i>	50.21%	51.13%	50.02%	49.47%
<i>bicg</i>	51.01%	50.58%	51.31%	51.13%
<i>deriche</i>	80.46%	76.96%	86.59%	77.85%

Table 7.7: Hoeffding Tree accuracy results for hypothetical optimised kernels. Leaf nodes do split trials on every 10 samples collected.

Model Name	Average Accuracy	Individual Runs		
All kernels	55.57%	59.63%	51.79%	55.27%
<i>adi</i>	72.67%	73.80%	68.80%	75.42%
<i>2mm</i>	76.31%	75.79%	75.88%	77.25%
<i>atax</i>	50.18%	49.47%	50.48%	50.58%
<i>bicg</i>	50.67%	51.13%	49.75%	51.13%
<i>deriche</i>	81.92%	82.64%	85.65%	77.47%

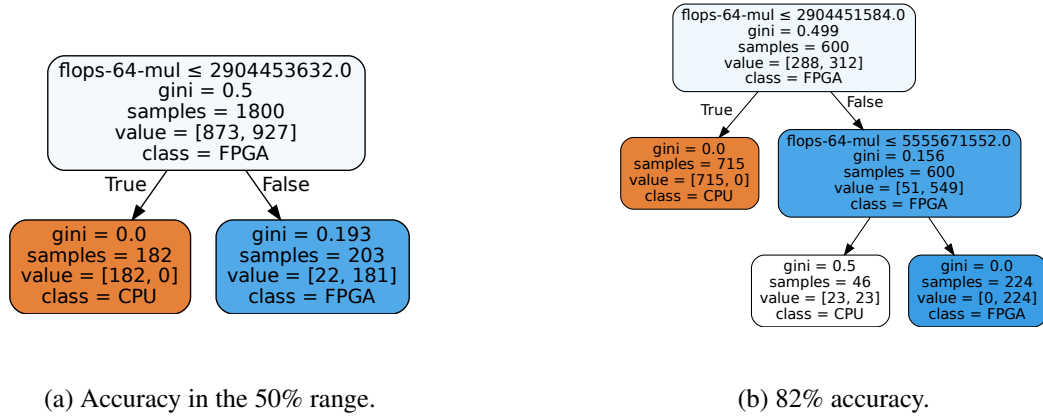


Figure 7.10: Visualisation of the decision tree models resulting from learning the dataset generated for a hypothetical $2mm$ kernel. Leaf nodes attempt to split at every 200 samples. Sample counts on each non-leaf node represent the state of the node when it was split.

For an arbitrary kernel with an arbitrary number of input parameters the methodology of extracting the 16 types of operation counts validates that it is possible to distil the parameter space complexity into a standardised set of arithmetic operation counts representing the computational load, resorting only to automated source analysis and transformations at compile time.

7.3.7 Hoeffding Tree Scheduler

In the previous experiment, using the hypothetical kernels dataset, the tree learned with every incoming sample. That does not represent a scheduler as one must be able to decide between execution platforms instead of asking the system to profile every single execution. In this final experiment, we present and evaluate our proposed portable Hoeffding Tree scheduling algorithm.

We used the following code to determine if the model should learn or infer on the n -th kernel execution request.

```

1 #define BINSIZE 2000
2 train = (n % BINSIZE) < (BINSIZE * (1 / (pow(2, 1 + uint(n / BINSIZE)))));

```

Listing 7.2: Expression used to reduce learning requirements over time.

This results in a exponential reduction of training over time. Training will occur with the first 1000 samples of the first 2000 sample group. In the second group, the scheduler will only use 500 to train, 250 in the third group, etc. Effectively, the scheduler has a warm-up period that diminishes over time.

The accuracy results of the scheduler can be seen in Table 7.8. The last column shows the percentage of samples from which learning was conducted instead of making a scheduling decision.

Again, the order in which the samples come in affects the results greatly. If the samples come in an order such that the tree is able of splitting early (or at all), the accuracy increases sharply.

Table 7.8: Scheduler accuracy results hypothetical optimised kernels. Leaf nodes do split trials on every 50 samples collected.

Kernel Name	Avg. Acc.	Individual Runs					Nr. Samples	Nr. Inferences	Training
All kernels	55.50%	51%	50%	63%	62%	51%	12761	10775	15,56%
<i>adi</i>	59.51%	50%	51%	50%	50%	96%	1981	981	50,48%
<i>2mm</i>	67.18%	93%	49%	50%	49%	95%	2185	1000	54,23%
<i>atax</i>	52.08%	51%	53%	52%	53%	52%	2169	1000	53,90%
<i>bicg</i>	50.80%	51%	52%	50%	50%	51%	2169	1000	53,90%
<i>deriche</i>	73.84%	75%	75%	81%	70%	69%	4257	2507	41,11%

To mitigate this problem a base tree model could be provided that is augmented and adjusted to a specific device on runtime.

As seen, we were able to validate the full flow use of *Tribble* and, through this model of computational load, validate that our Hoeffding Tree scheduler can, under certain conditions, effectively choose between kernel implementations. A great dependability on the order of sample input is present. There is also a sharp decrease in accuracy when considering models for individual kernels vs. the generic model with all samples (*All kernels*). Larger, more comprehensive datasets may provide additional insights into the matter. Different bin sizes and methods for determining the warm-up period should also be explored as well as adding an additional feature to the feature vector to identify each kernel. Finally, with minor modifications to *Tribble*'s code generation, any of the two approaches could be used (individual vs generalised models) to achieve the best results.

Chapter 8

Conclusions

With the move to increasingly heterogeneous architectures, we have seen that the effort required to create software for these complex architectures is immense as most tasks such as memory management and task allocation require manual tuning and expert knowledge of the underlying architectural details.

We have identified a few shortcomings in the state-of-the-art tools aiming to mitigate this issue (Section 3.1). Several are not transparent and require that programs be developed around their APIs and programming models. Other dependencies such as on the LLVM Intermediate Representation (LLVM-IR) for the analysis of the code and injection of schedulers makes it so that a developer is forced to compile their program using the author’s compiler. In many situations, this is not possible. The Xilinx toolchain, for example, uses GNU C Compiler (GCC) as the compiler for the host program and a number of scripts to configure it. Program portability was always impaired by requirements of offline retraining of ML models or outright code modifications. In general, all tools required the developer to adapt their workflow to them instead of offering a totally transparent workflow.

We have demonstrated the capability and validity of the end-to-end compilation flow of *Tribble* as a means of transparently generating an accelerated version of any C/C++ program targeting a heterogeneous architecture. Experimental evaluation was conducted of *Tribble* and of our Hoeffding Tree implementation on a ZCU102 board. These experiments showed that our non-optimised Hoeffding Tree HLS implementation can achieve speedups of 8.3× in complex inference tasks over a pure software execution. They also demonstrate that HLS is no substitute for expert manipulation of code.

In line with our goal of code portability, we propose a scheduling algorithm based on our Hoeffding Tree implementation and show that it is possible to use arithmetic operation counts as a means of constructing a scheduling model capable of selecting the optimum platform for task execution with an accuracy of up to 96%.

8.1 Contributions

The contributions of this work are as follows:

- A source-to-source compilation framework capable of automatically and transparently generate accelerated programs targeting [CPU+FPGA](#) heterogeneous environments.
- Experimental evaluation of the capabilities of the *Tribble* framework.
- A generic, template-based C/C++ implementation of the Hoeffding Tree classifier as per Lin et al. [24], but that is suited for [HLS](#) and can be used on a [CPU](#) or [FPGA](#).
- Functional validation of the tree implementation through software execution, and post-synthesis onto a Xilinx ZCU102 development board.
- Experimental evaluation of memory requirements of the tree object as a function of template parameters.
- Experimental evaluation of [FPGA](#) resource requirements and execution time of the synthesised training and inference method as a function of template parameters for our Hoeffding Tree implementation.
- Two conference publications (see Appendix [A](#))

8.2 Future Work

1. Fix the limitations detailed in sections [5.11](#) and [6.3](#).
2. Currently our proposed scheduler only takes into account arithmetic operation counts. Exploration of the effect that counting data access operations and incorporating them into the model is a path worth pursuing.
3. The *Tribble* profiling of the [FPGA](#) task execution time is for the total time taken to complete the task. This could be divided into the data transfer and compute portions to support more complex models.
4. [DPR](#) is mentioned in chapter [1](#) as a major enabler for the compute model we try to explore. The scheduler proposed in this document does not take into account reconfiguration times due to [DPR](#) or if the kernel is already available.
5. Explore if the operation counts measured via the [AST](#) differ heavily from the number obtained through the [IR](#). To do this each scope could be compiled individually and compared to the current results.
6. Recent Xilinx and Intel tools support the synthesis of ONNX models to heavily optimised IP blocks. Our Hoeffding Tree implementation is exportable to the SciKit Learn format.

From there, the *skl2onnx* Python package can be used to convert it to the ONNX format. Comparisons can be made between the inference performance of our [HLS](#)-generated models to the ones from ONNX.

7. Characterisation of the overhead introduced by the *Tribble* framework and scheduler.
8. Automatic identification of computationally heavy workloads, amenable for acceleration on [FPGAs](#), to reduce *Tribble*'s the dependency on the user to be able to correctly identify usable kernels.
9. Explore Regression Decision Trees and non-Decision Tree-based models as scheduler options.
10. Explore other feature sets for model training.
11. Expand *Tribble* to support regression models and more than two target devices.

Appendix A

Publications

Table of Contents

1. LATTE'21 - The workshop on Languages, Tools, and Techniques for accelerator Design is a workshop embedded in the Architectural Support for Programming Languages and Operating Systems conference. ASPLOS is rated as a CORE A* conference.
2. ISCAS 2022 - This paper has been submitted to the IEEE International Symposium on Circuits and Systems and is currently being peer-reviewed.

A Position on Transparent Reconfigurable Systems

Luís M. Sousa
lm.sousa@fe.up.pt
Faculty of Engineering, University of
Porto
Porto, Portugal

Nuno Paulino
nmcp@fe.up.pt
INESC-TEC and Faculty of
Engineering, University of Porto
Porto, Portugal

João Canas Ferreira
jcf@fe.up.pt
INESC-TEC and Faculty of
Engineering, University of Porto
Porto, Portugal

ABSTRACT

With the ever more pressing issue arising from the phenomenon known as the *death* or *slowdown* of Moore’s Law and the Dennard Scaling, compute performance has not been increasing at the rate the industry had been accustomed to over the decades [7]. This has prompted a shift from mostly homogeneous compute architectures to increasingly heterogeneous ones [1]. As these systems become increasingly complex, manual tuning and management of these heterogeneous resources becomes unfeasible. In this paper, we propose a runtime mechanism for automatic reconfigurable resource management that will enable a hypothetical flow for combined hardware and software compilation.

1 INTRODUCTION

An application’s performance can be maximised if it is executed on specialised hardware, both from a time and power consumption perspective. As customers demand higher performance and functionality, complexity and development time tend to increase. Furthermore, since silicon area is expensive [2], the use of Hardware Accelerators (HwAs) is only justified under certain conditions. Firstly, that they are used frequently to justify the design time and area expended. Secondly, that their workload is both well defined and amenable for parallelization, so that performance benefits can be maximised.

A task is only worth using an accelerator if the time it takes to be completed on such an accelerator is lower than the time it would take on general-purpose hardware. The time to transfer the data to the accelerator must be taken into account. Consider an embedded application with N functions for which accelerator circuits have been created. At run-time, the functions are called with arbitrary arguments, potentially in an unknown order. If the device harbouring the HwAs is incapable of implementing all N accelerators concurrently, some sort of management must be made in order to make available the most valuable subset of HwAs at any given time.

Traditionally, the developer of a heterogeneous system must understand the algorithmic component of the application, design the appropriate hardware to accelerate candidate regions (e.g. bottlenecks or good parallelization opportunities), and then must schedule workloads onto the different components, and synchronise their behaviour [12, 15].

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
LATTE ’21, April 15, 2021, Virtual, Earth
© 2021 Copyright held by the owner/author(s).

The hardware design aspect has been addressed, in part by new tools such as High-Level Synthesis (HLS) which can be used to automatically generate circuit descriptions from functions written in C/C++ code [8, 14]. This enables simpler and lower cost development workflows. For certain execution models such as Field-Programmable-Gate-Array (FPGA) based server boards, HLS also provides Application Programming Interface (API) level integration of the resulting components.

Although FPGA devices lack the advantages of full-custom, i.e. Application Specific Integrated Circuit (ASIC) implementations [9], they introduce new paradigms to these heterogeneous systems by allowing the application to define custom circuitry to be implemented.

This capability for reconfiguration allows for a single underlying chip to be used to implement the HwAs required by a specific application. Additionally, algorithms are subject to change due to performance reasons, different application needs, or bug fixes. As FPGAs are reconfigurable, the cost-cutting is multiplied as no new devices need to be fabricated and deployed.

Another advantage of FPGAs for application-specific HwA design is the unique ability of *hot-swapping* [6] accelerator circuits at run-time, which is referred to as Dynamic Partial Reconfiguration (DPR) [13]. This feature, which has not seen widespread adoption in real-world applications, allows for a targeted FPGA area to be reconfigured while the device is in operation, allowing for the same resources to implement multiple functions in a time-multiplexed manner. (Figure 1) Allowing an application to hot-swap accelerators instead of solely dealing with a fixed, predefined set, implemented at boot-time, greatly expands on the possibilities of using FPGA devices as platforms for implementing accelerators. Those that are not used concurrently can be swapped out as needed freeing up space for others, reducing the total area of re-configurable fabric required to implement the complete set of HwAs of a specific application. By allowing time-multiplexed use of limited silicon resources, smaller and less expensive devices can be utilised.

Given this context, developers that wish to exploit heterogeneity in the context of FPGAs for the embedded domain, while also relying on their DPR capability for better silicon area usage, must therefore manually perform hardware/software partitioning, hardware design and testing, and runtime management of reconfigurable slots, both spatially and temporally.

2 PROPOSED SOLUTION

Our work focuses on the implementation of a runtime resource management mechanism based on decision trees. We envision this as part of compiler-based hardware/software partitioning approaches

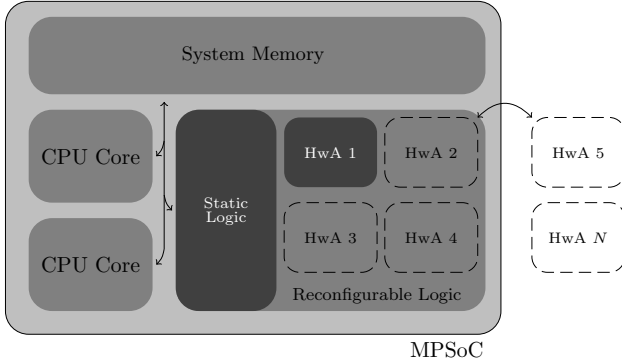


Figure 1: Proposed Architecture

previously proposed in the state-of-the-art, which can be summarised as 1) identifying segmented regions of code amenable for acceleration, 2) generating circuits for those regions without developer intervention, and 3) inserting invocations to the custom hardware. We aim to augment the accelerator invocation runtime management capabilities. Namely, 1) determining if a given accelerated function can indeed benefit from hardware invocation, versus its software counterpart, based on runtime function input parameters, and 2) falling back to software execution based on available accelerator slots.

This would determine if it is advantageous to implement an accelerator for a specific task execution and offload it to a DPR-based slot or execute that task on the general-purpose hardware.

Previous work has been done with this concept. However, the approaches used require time-consuming offline profiling and training of the used models [3, 4, 11] not consistent with compiler flows. These can provide very good offloading decisions on a system ensuring great performance but at the sacrifice of portability. The models developed for one particular system are not applicable if any of the components, or their operating frequency are changed, even when maintaining the same Instruction Set Architecture (ISA). Any change in system configuration will require profiling of the new system and retraining.

Our proposal, intended to automate the use of HLS and DPR transparently, requires no offline profiling and therefore no expended time dealing with changes to the code base. A custom decision engine can be trained on a per-device level that provides decisions tailored to the application running on such a device by employing online Decision Tree (DT) learning [5] fed by data such as the values of the arguments of the functions that are being called, the size of the data they will operate on, an estimate of the arithmetic intensity obtained through static analysis, the order in which the functions are called, whether or not the corresponding HwA is already loaded into the FPGA and the temporal overhead of loading the HwA via DPR (Algorithm 1).

The process starts by using the DT to decide if either software or hardware execution should be used to target each individual function call. If the DT's confidence level in its decision is above a user-defined threshold, the chosen component will execute the function and return the result. On the other hand, if the confidence

Algorithm 1: Runtime Decision Engine

```

Initialisation;
Use the parameters of the function call in DT.
Let  $\sigma$  be the confidence in the choice made by the DT.
if  $\sigma > \text{confidence threshold}$  then
    | Execute function on the preferred platform.
else
    | Start execution on CPU.
    | Start timer.
    | Use DPR to load the HwA (if needed).
    | Copy input data to HwA.
    | Start execution on HwA.
end
if CPU or HwA is finished then
    | Halt execution of slower platform.
    | Copy result from HwA (if available).
    | Stop timer.
    | Train DT using the winner.
else
end

```

level in the decision is low, the function is executed concurrently on both Central Processing Unit (CPU) and HwA. Execution is halted once either concurrent execution (i.e., software and hardware) terminates. The result is used to train the DT, increasing its confidence level in that region of the problem space.

The Hoeffding Tree algorithm [5] is the method we propose to use in this Engine. It guarantees an asymptotically identical result when compared to batch learners, assuming that the data distribution does not change over time. Furthermore, several improvements to the original algorithm have also been proposed to make it more efficient when executed on an FPGA architecture [10].

By compiling the information of all the leaf nodes on the DT, a list of the most used HwAs can be obtained. These can be kept preloaded in the FPGA fabric to minimise DPR overhead. Take a pair of identical devices (A & B) deployed in different conditions. Device A may use HwA 1 more often than device B that prefers HwA 2 due to the conditions surrounding the devices. By keeping HwA 1 preloaded on device A better performance can be achieved by cutting the reconfiguration time. Device A may also have a better CPU than device B. This can mean that, for particular scenarios, device A may prefer CPU execution where device B will use HwA 3.

We propose to explore the implementation of such decision algorithms, firstly through software implementations on-chip running in an auxiliary processor, and then via hardware implementations of the DT. Leveraging technologies such as HLS and DPR, we thus aim to increase the abstraction level given to programmers for the use of re-configurable resources by shifting these development concerns towards the compiler.

REFERENCES

- [1] Abderazak Ben Abdallah. 2017. Heterogeneous Computing: An Emerging Paradigm of Embedded Systems Design. In *Computational Frameworks: Systems, Models and Applications*. Elsevier, 61–93. <https://doi.org/10.1016/B978-1-78548-256-4.50003-X>

- [2] Mark T. Bohr and Ian A. Young. 2017. CMOS Scaling Trends and Beyond. *IEEE Micro* 37, 6 (2017), 20–29. <https://doi.org/10.1109/MM.2017.4241347>
- [3] Usman Dastgeer, Lu Li, and Christoph Kessler. 2013. Adaptive implementation selection in the SkePU skeleton programming library. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 8299 LNCS. Springer, Berlin, Heidelberg, 170–183. https://doi.org/10.1007/978-3-642-45293-2_13
- [4] D. del Rio Astorga, Manuel F. Dolz, Javier Fernandez, and Javier Garcia Blas. 2019. Hybrid static–dynamic selection of implementation alternatives in heterogeneous environments. *Journal of Supercomputing* 75, 8 (8 2019), 4098–4113. <https://doi.org/10.1007/s11227-017-2147-y>
- [5] Pedro Domingos and Geoff Hulten. 2000. Mining high-speed data streams. In *Proceeding of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Association for Computing Machinery (ACM), New York, New York, USA, 71–80. <https://doi.org/10.1145/347090.347107>
- [6] John L. Hennessy and David A. Patterson. 2019. *Computer architecture: a quantitative approach* (sixth ed.). Morgan Kaufmann.
- [7] Mark Horowitz. 2014. 1.1 Computing’s energy problem (and what we can do about it). In *Digest of Technical Papers - IEEE International Solid-State Circuits Conference*, Vol. 57. 10–14. <https://doi.org/10.1109/ISSCC.2014.6757323>
- [8] Intel. 2020. High-Level Synthesis Compiler. <https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html>
- [9] Ian Kuon and Jonathan Rose. 2007. Measuring the gap between FPGAs and ASICs. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 26. 203–215. <https://doi.org/10.1109/TCAD.2006.884574>
- [10] Zhe Lin, Sharad Sinha, and Wei Zhang. 2019. Towards efficient and scalable acceleration of online decision tree learning on FPGA. In *Proceedings - 27th IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM 2019*. Institute of Electrical and Electronics Engineers Inc., 172–180. <https://doi.org/10.1109/FCCM.2019.00032>
- [11] William F. Ogilvie, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2015. Fast automatic heuristic construction using active learning. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 8967. Springer Verlag, 146–160. https://doi.org/10.1007/978-3-319-17473-0_10
- [12] Gavin Vaz, Heinrich Riebler, Tobias Kenter, and Christian Plessl. 2014. Deferring accelerator offloading decisions to application runtime. In *2014 International Conference on ReConfigurable Computing and FPGAs (ReConFig14)*. IEEE, Cancun, Mexico, 1–8. <https://doi.org/10.1109/ReConFig.2014.7032509>
- [13] Xilinx Inc. 2020. *Dynamic Function eXchange - Vivado Design Suite User Guide*. Technical Report. Online. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug909-vivado-partial-reconfiguration.pdf
- [14] Xilinx Inc. 2020. *Vivado High-Level Synthesis*. Technical Report. Online. <https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html>
- [15] Gina Yuan, Shoumik Palkar, Deepak Narayanan, and Matei Zaharia. 2020. Offload annotations: Bringing heterogeneous computing to existing libraries and workloads. In *Proceedings of the 2020 USENIX Annual Technical Conference, ATC 2020*. 293–306. <https://www.usenix.org/conference/atc20/presentation/yuan>

A Flexible HLS Hoeffding Tree Implementation for Runtime Learning on FPGA ¹

Abstract—Decision trees are often preferred when implementing Machine Learning in embedded systems for their simplicity and scalability. Hoeffding Trees are a type of Decision Trees that take advantage of the Hoeffding Bound to allow them to learn patterns in data without having to continuously store the data samples for future reprocessing. This makes them especially suitable for deployment on embedded devices. In this work we highlight features of an HLS implementation of the Hoeffding Tree. The implementation parameters include the feature size of the samples (D), the number of output classes (K), and the maximum number of nodes to which the tree is allowed to grow (N_d). We target a Xilinx MPSoC ZCU102, and evaluate: the design's resource requirements and clock frequency for different numbers of classes and feature size, the execution time on several synthetic datasets of varying sample sizes (N), number of output classes and the execution time and accuracy for two datasets from UCI. For a problem size of D3, K5, and N40000, a single decision tree operating at 103MHz is capable of 8.3× faster inference than the 1.2GHz ARM Cortex-A53 core. Compared to a reference implementation of the Hoeffding tree, we achieve comparable classification accuracy for the UCI datasets.

Index Terms—component, formatting, style, styling, insert

I. INTRODUCTION

With the rise of edge computing, FPGA vendors have been releasing and marketing CPU+FPGA SOCs as the ideal solution for this domain. As edge devices are often specialised for a single task in a constrained environment, it is advantageous to build dedicated hardware to improve performance and energy efficiency. FPGAs offer the advantage of targeted hardware without losing the ability to adapt the platform to changes (e.g., security updates), while being more efficient than a pure software solution.

As High Level Synthesis (HLS) matures [1], it becomes a more attractive approach to creating efficient high-performance accelerators for FPGA devices.

Machine Learning (ML) algorithms are a prime candidate for acceleration at the edge, but their computational requirements exceed the capabilities of many embedded devices. Inference at the edge is a problem being addressed by many works, but training at the edge still faces hurdles to adoption despite its clear benefits. In the field of Decision Trees (DTs), many algorithms are incompatible with devices of this class due to memory constraints. ID3 [2], and derivatives such as C4.5 and C5.0 require the entire training dataset be present in memory for training. Incremental learning algorithms as ID5 [3], ID5R [4] and ITI [5] do allow for ongoing learning from streaming data but store the dataset samples within the tree.

Hoeffding Trees [6] are incremental learning trees, which are more suitable for embedded scenarios because they have the following advantages: They asymptotically guarantee the

same classification as traditional batch learners, and they store information about the distribution of samples statistically rather than the samples themselves, which drastically reduces memory requirements, especially for large datasets.

In this work, we present a flexible C/C++ HLS implementation of a Hoeffding Tree variant tailored for use in FPGAs, originally proposed by Lin et al. [7]. Their work built on an earlier variant in which the storage of the statistical data of the sampling distribution of the original Hoeffding Tree was replaced by a Gaussian approximation [8]. Lin et al. replace this approximation with quantile estimation using asymmetric sigmum functions [9]. The result is a larger memory footprint but a reduction in computational requirements, while achieving similar results. Since it is implemented in Verilog, the applicability of the implementation is limited to circuit synthesis, e.g. for FPGA. By using HLS, an implementation can be created that is equally suitable for CPU and FPGA.

The contributions of this work are as follows:

- A generic, template-based C/C++ implementation of the Hoeffding Tree classifier as per Lin et al. [7], but that is suited for HLS.
- Functional validation of the implementation through software execution, and post-synthesis onto a Xilinx ZCU102 development board.
- Experimental evaluation of memory requirements of the tree object as a function of template parameters.
- Experimental evaluation of FPGA resource requirements and execution time of the synthesised training and inference method as a function of template parameters.

II. HLS Hoeffding Tree Implementation

A decision tree is a type of machine learning algorithm used either for classification or regression. A decision tree performs sequential binary decisions over an incoming vector of features, and a classification is computed when a leaf node is reached. During training, leaf nodes are added to the tree based on a splitting criteria, which separates the data into two regions at every tree junction. A Hoeffding tree is a type of decision tree where the criteria is the Hoeffding bound, shown in Equation 1. The tree performs learning and inference by relying on a property of the Hoeffding bound that guarantees that best splitting point is chosen. If a gain function G , is to be maximised, then given $G(X)$ and $G(Y)$ (X and Y being the attributes that generate the highest and second highest values of G) if $G(X) - G(Y) > \epsilon$ then the Hoeffding bound guarantees that with probability $1 - \delta$ X is the best attribute to split on. R represents the range of the attributes e N the number of samples on a node.

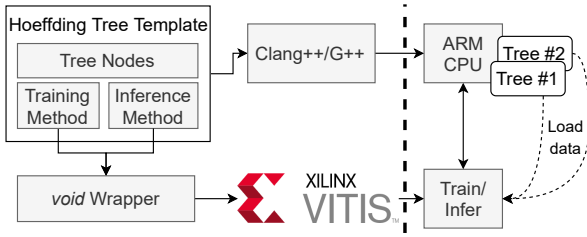


Fig. 1. Software and hardware architecture of the Hoeffding Tree implementation; the training and inference kernels are shared by multiple tree objects

$$\varepsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2N}} \quad (1)$$

Over other criteria, the Hoeffding bound has two characteristics: it allows for online incremental learning and growth of the tree which asymptotically tends towards the results provided by batch learners, and is independent of the probability distribution of the data sampling. The Hoeffding tree allows for continuous learning and node splitting for a potentially infinite (e.g. streaming applications) number of samples [6].

FPGAs have been intensively studied for decision tree implementations, as a tree structure maps efficiently to specialised hardware. In conjunction with other optimisations, decision trees in FPGAs have been shown to outperform CPU and GPU solutions [10]. Lin et al. [7] demonstrate speedups of up to 1500x for an RTL implementation of the Hoeffding tree versus a 2.6GHz processor. Our aim is to explore a higher abstraction level via HLS, providing greater applicability features, while evaluating the attainable performance.

We implemented the tree as a C++ class template. The parameters include the maximum number of nodes in the tree, the feature size, and the floating-point precision. The class contains the training and inference methods which are synthesised to hardware. At runtime, the C++ tree object can be manipulated in software, and passed as an argument to the training/inference method, as summarised in Figure 1.

This allows for instantiation of several tree objects in memory (with different template parameters if desired). Trees with the same template parameters can be processed by the same synthesised circuit. Since the functions can also be invoked in software, this means that training or inference can be dynamically partitioned based on which device performs better for either task, as a function of the tree parameters. This also means that if FPGA is occupied processing a tree object, other trees can be evaluated via software without the need for a blocking wait.

Finally, evaluation of multiple trees is possible by either a combination of software and hardware invocations, by deploying multiple instances of the hardware kernel, or by time-multiplexing a single hardware kernel (as explained below). Either case allows for the possibility of arbitrary runtime tree ensembles. This evaluation is currently future work.

The Xilinx Vitis HLS flow enforces an OpenCL model for kernel invocation. The implemented kernel, `krnl_Tree`, receives 4 arguments. A `HoeffdingTree` object as men-

tioned, an array of samples, an array of output classifications, and the size of these arrays.

In this model, a large overhead penalty would occur for invocations with a single sample, due to the data transfer time. A practical application of the kernel design could be, e.g., in the sensor domain, where the tree could continuously sample fused data from multiple sensors (i.e., multiple attributes) without processor intervention, avoiding transfer overheads. Alternatively, streaming samples can be accumulated until a sufficiently large number is held that mitigates this overhead. This does not mean that the tree behaves as a batch learner, as one sample is processed per each *infer-then-train* step.

Inference on an incremental learning decision tree cannot be easily parallelised as the model changes and evolves with every training sample that arrives. This restricts the pipeline to dealing with one sample at a time, sequentially. The sample structure contains information about whether it should be used for training purposes or only for inference. Thus, as the kernel loops through the sample array, it executes either the `train` or `infer` method of the tree object accordingly. The results are placed in the output data structure.

The OpenCL API allows for fine-grained control of how these arguments are passed to the kernels, each argument being a separate buffer with persistent storage. Thus, trees can be transferred to FPGA memory once, and not retrieved between executions of the kernels. With this mechanism, a tree object can reside in memory while only new samples are transferred in, and the model can be retrieved in a final stage.

Conversely, the samples themselves may remain in memory, and trees freely exchanged. This is one strategy for the construction of tree ensembles mentioned previously. Trees can reuse the same kernel instance via time-multiplexing, or by concurrent instantiation of several copies of `krnl_Tree`. In either case, the same read-only sample buffer can be assigned to all trees, thus significantly reducing overhead and preventing data duplication. For brevity, the evaluation of ensembles is out of the scope of this paper.

III. EXPERIMENTAL EVALUATION

We performed the following experiments: evaluated the resource utilisation of a single synthesised tree for a range of values for the feature size and number of classes; evaluated the training and inference time of a single tree in hardware, versus the ARM CPU, for several synthetic clustering datasets (varying number of point, clusters, and feature size); evaluated the classification accuracy and execution time of a single tree for UCI's Bank and Covertype datasets.

A. Resource Utilisation

Table I presents various configurations of the kernel, tailored for datasets of different dimensions (D), with different number of classes (K), number of samples (N) and max number of nodes (Nd). The purpose is to determine the effect of these parameters on FPGA resource utilisation. As expected, parameter N has no effect on resource utilisation as samples cannot be processed in parallel.

The feature size and the number of classes result in an increase in resource usage. This is due to the highly sequential nature of the generated kernel, which also explains why the performance of this kernel on training tasks is poor compared to the CPU. This overall advantage is less surprising when considered in the context of an 11-fold CPU advantage in clock speed. Current HLS tools cannot automatically parallelize sequential code. Without hardware design expertise in order to optimise the design, the implementation will be far from optimal. In our implementation, we still believe that further parallelization can be achieved even within a single tree, through inner loop unrolling or memory partitioning.

One interesting result is that of the kernel's operating frequency. It remains unchanged for all configurations. Looking deeper into the cause of this phenomenon, one finds that the bottleneck is the sorting of a sample down from the root node to the appropriate leaf node. This sequential operation also prevents the kernel from being pipelined.

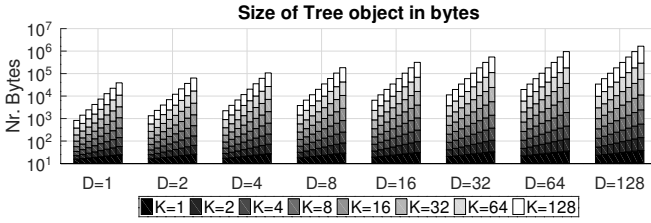


Fig. 2. Size of Tree objects in bytes for Nd, D and K. Each bar in every grouping, depicts a tree with a max number of nodes from 2^0 to 2^7 .

B. Performance

These results were obtained by feeding the tree with datasets of K clusters in a D dimensional spaces, constituted of N points. For these experimental runs, we will have the entire dataset transferred in a single operation to the FPGA's memory.

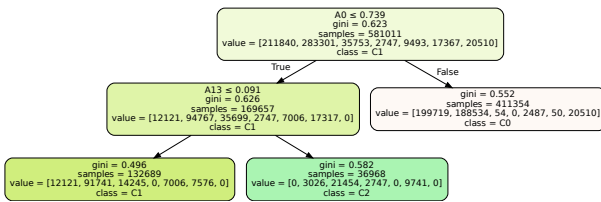


Fig. 3. Illustrative visualisation of tree model derived from UCI Covertypes dataset. The tree was only allowed to grow to 5 nodes.

Looking at the first four rows of Table II (D=3) it can be observed that for a 3-dimensional dataset, regardless of the bundle size, the ARM CPU in the ZCU102 SoC significantly outperforms the FPGA implementation in both the training and inference tasks. Also, the performance gap between both implementations grows with the number of samples processed. This indicates that the kernel is slower, per iteration, than the pure software solution. Regarding the last four rows of Table II (D=100), the ARM CPU still outperforms the FPGA kernel in training. However, it does it with a lower margin

and one that does not appear to grow with the added number of samples. On the inference task with this larger dataset, the FPGA outperforms the ARM processor by 8.3×.

Table III presents benchmarks of two of the UCI datasets used by Lin et al. [7]. The same tree parameters were used ($\delta = 0.001$, $\lambda = 0.01$, $\tau = 0.05$, $n_{min} = 200$, $n_{pt} = 10$, $n_{quantiles} = 16$, $Nd = 2047$), with one being of special relevance: Nd (maximum number of nodes). A significant slowdown occurred. With the increased number of nodes, the sequential tree traversal algorithm increases in length. Our HLS implementation achieves comparable accuracy for *Bank*, although the performance for *Covertypes* is inferior. Lin et al. [7] reports 89.30% and 72.51%, respectively. We believe a difference in calculation precision between the CPU and FPGA caused the degradation, despite the use of 32-bit floating point data types for both devices.

IV. RELATED WORK

Kulaga et al. [11] present an HLS decision tree ensemble solution for inference tasks. The results achieved are competitive regarding performance when compared to the ARM core present in the tested SoC. However, the design is highly dependent on the number of trees and corresponding depths, as a change in ensemble parameters requires re-tuning multiple pragmas. As we have also seen, an unavoidable sequential portion of the algorithm is the sample sorting through the tree structure. Unlike our approach, the number of trees in an ensemble is hardcoded into the synthesised kernel. In contrast, by having one or more synthesised training/inference methods (for different hyper-parameters), we can deploy N instances of such circuits and process a runtime allocated number of trees.

As previously stated, the work on this paper builds on Lin et al. [7] work. However, their implementation is closed-source and done in Verilog, which excludes native execution on CPUs. Also, as the work was developed for a datacenter-class FPGA device, the implementation is very resource intensive and thus not suitable for small devices such as the ones used on embedded systems.

InAccel² provides an HLS implementation of the XGBoost learning algorithm, which is also based on decision trees. For a dataset of 65k points, 5 classes, and 128 features, the training time is 2.7 seconds. This is significantly faster than our performance for similarly sized datasets, but InAccel's implementation targets server-grade FPGA accelerator boards (including multi-board setups), while we target the embedded domain. However, the potential for HLS FPGA acceleration of decision tree algorithms is demonstrated, given expert optimisation of the code for HLS.

V. CONCLUSIONS

We presented a flexible and scalable implementation of a Hoeffding Tree compatible with HLS tools³. We performed a functional validation of the tree design, against software execution, by implementation on chip on a Xilinx ZCU102.

²InAccel, 2019, XGBoost Exact Updater IP core, <https://github.com/inacel/xgboost>

³Omitted for blind review

TABLE I
N, D, K AND ND EFFECTS ON FPGA RESOURCE UTILISATION

Nodes	100	100	100	1000	100	100	100	1000
K	5	5	10	5	5	5	10	5
D	3	100	3	3	3	100	3	3
N	40k	40k	40k	40k	500k	500k	500k	500k
LUT	23304 (8.6%)	20567 (7.6%)	23776 (8.8%)	24351 (9.0%)	23304 (8.6%)	20567 (7.6%)	23776 (8.8%)	24351 (9.0%)
LUTRAM	1395 (1.0%)	1179 (0.8%)	1399 (1.0%)	1397 (1.0%)	1395 (1.0%)	1179 (0.8%)	1399 (1.0%)	1397 (1.0%)
FF	35682 (6.6%)	29775 (5.5%)	36374 (6.7%)	36336 (6.7%)	35682 (6.6%)	29775 (5.5%)	36374 (6.7%)	36336 (6.7%)
BRAM	12 (1.3%)	9.5 (1.0%)	12 (1.3%)	12 (1.3%)	12 (1.3%)	9.5 (1.0%)	12 (1.3%)	12 (1.3%)
DSP	23 (0.9%)	25 (1.0%)	25 (1.0%)	25 (1.0%)	23 (0.9%)	25 (1.0%)	25 (1.0%)	25 (1.0%)
BUFG	13 (3.2%)	13 (3.2%)	13 (3.2%)	13 (3.2%)	13 (3.2%)	13 (3.2%)	13 (3.2%)	13 (3.2%)
MMCM	1 (25.0%)	1 (25.0%)	1 (25.0%)	1 (25.0%)	1 (25.0%)	1 (25.0%)	1 (25.0%)	1 (25.0%)
Freq. (MHz)	103.6	103.6	103.6	103.6	103.6	103.6	103.6	103.6

TABLE II
TRAINING AND INFERENCE TIMES FOR FOUR SYNTHETIC CLUSTERING DATASETS, FOR THE ARM CPU (1.2GHZ) AND THE FPGA (103MHZ)

K	D	N	Task	ARM CPU	FPGA	Speedup
3	40k		Training	207 ms	1,990 ms	0.10×
			Inference	151 ms	462 ms	0.33×
	500k		Training	2,983 ms	30,933 ms	0.10×
			Inference	2,260 ms	11,442 ms	0.20×
	100	40k	Training	6,028 ms	51,648 ms	0.12×
			Inference	3,924 ms	469 ms	8.37×
5	500k		Training	75,763 ms	651,775 ms	0.12×
			Inference	49,495 ms	11,494 ms	4.31×

TABLE III
TRAINING TIME AND ACCURACY (ACC.) FOR COVERTYPE AND BANK DATASETS, FOR THE ARM CPU (1.2GHZ) AND THE FPGA (103MHZ)

	ARM CPU		FPGA		Speedup
	Acc.	Time	Acc.	Time	
Bank	88.3%	202 ms	88.3%	8,525 ms	0.02×
Coverttype	72.2%	9,712 ms	63.7%	374,600 ms	0.03×

We provide a evaluation of the design’s resource usage for multiple template parameter values (i.e., maximum tree size, number of sample attributes, number of clusters, and number of dataset samples), as well as execution time versus an ARM Cortex-A53 processor. The resource requirements of the tree do not scale significantly with problem size, although further HLS optimisations such as unrolling remain unexplored. Even so, we outperform the ARM by 8.3x times for largest dataset for the inference task, while being 8.6x slower during training. As future work, we envision the use of tree ensembles, and the partitioning of training and inference task between software and hardware based on problem size.

ACKNOWLEDGMENTS

[omitted for blind review]

REFERENCES

- [1] Xilinx Inc., “Vivado High-Level Synthesis,” tech. rep., Online, 2020.
- [2] J. R. Quinlan, “Learning Efficient Classification Procedures and Their Application to Chess End Games,” *Machine Learning*, pp. 463–482, 1983.
- [3] P. E. Utgoff, “ID5: An Incremental ID3,” in *Machine Learning Proceedings 1988*, pp. 107–120, Elsevier, 1 1988.
- [4] P. E. Utgoff, “Improved Training Via Incremental Learning,” in *Proceedings of the Sixth International Workshop on Machine Learning*, pp. 362–365, Elsevier, 1 1989.
- [5] P. E. Utgoff, N. C. Berkman, and J. A. Clouse, “Decision Tree Induction Based on Efficient Tree Restructuring,” *Machine Learning 1997 29:1*, vol. 29, no. 1, pp. 5–44, 1997.
- [6] P. Domingos and G. Hulten, “Mining high-speed data streams,” in *Proceeding of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, (New York, New York, USA), pp. 71–80, Association for Computing Machinery (ACM), 2000.
- [7] Z. Lin, S. Sinha, and W. Zhang, “Towards Efficient and Scalable Acceleration of Online Decision Tree Learning on FPGA,” in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 172–180, IEEE, 4 2019.
- [8] B. Pfahringer, G. Holmes, and R. Kirkby, “Handling Numeric Attributes in Hoeffding Trees,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5012 LNAI, pp. 296–307, 2008.
- [9] A. Althoff and R. Kastner, “An Architecture for Learning Stream Distributions with Application to RNG Testing,” *Proceedings - Design Automation Conference*, vol. Part 128280, 6 2017.
- [10] M. Barbareschi, S. Barone, and N. Mazzocca, “Advancing synthesis of decision tree-based multiple classifier systems: an approximate computing case study,” *Knowledge and Information Systems 2021 63:6*, vol. 63, pp. 1577–1596, 4 2021.
- [11] R. Kułaga and M. Gorgoń, “FPGA Implementation of Decision Trees and Tree Ensembles for Character Recognition in Vivado Hls,” *Image Processing & Communications*, vol. 19, pp. 71–82, 9 2014.

Appendix B

adi Code Transformations

B.1 Original Code

```
1  | -_HLS_graphs
2  | -_HLS_reports
3  | -CxxIgnored
4  |   | -OCL_Helpers.hpp
5  |   | -OCL_Helpers.cpp
6  | -CxxSource
7  |   | -adi.cpp
8  |   | -adi.h
9  |   | -polybench.c
10 |   | -polybench.h
11 | -CxxTemplates
12 |   | -OCLH_main.cpp
```

Listing B.1: Original Folder Structure

B.1.1 CxxTemplates/OCLH_main.cpp

```
1  #include "../CxxIgnored/OCL_Helpers.hpp"
2
3  cl::Context context;
4  cl::CommandQueue queue;
5  cl::Program program;
6
7  void scheduler(uint64_t ops[], bool &executeOnCPU, bool &executeOnFPGA,
8                bool &lowConfidence) {
9      executeOnCPU = true;
10     executeOnFPGA = true;
11     lowConfidence = true;
```



```

12 }
13
14 void train(uint ops[], bool CPUwonFPGALost) { return; }
15
16 #pragma clava ocl_insert_globals
17 // cl::Kernel krnl_scheduler;
18
19 int main_template(int argc, char *argv[]) {
20
21     static const std::string platformName = "Xilinx";
22
23     OCLH::getConfig(argv[1], platformName, context, queue, program);
24
25 #pragma clava ocl_insert_kernel_initializations
26     // krnl_scheduler = OCLH::getKernel("scheduler", program);
27 }

```

Listing B.2: Original CxxTemplates/OCLH_main.cpp

B.1.2 CxxSource/adi.cpp

```

1 /**
2  * This version is stamped on May 10, 2016
3  *
4  * Contact:
5  *   Louis-Noel Pouchet <pouchet.ohio-state.edu>
6  *   Tomofumi Yuki <tomofumi.yuki.fr>
7  *
8  * Web address: http://polybench.sourceforge.net
9  */
10 /* adi.c: this file is part of PolyBench/C */
11
12 #include <math.h>
13 #include <stdio.h>
14 #include <string.h>
15 #include <unistd.h>
16
17 #define EXTRALARGE_DATASET
18 #define DATA_TYPE_IS_FLOAT
19
20 /* Include polybench common header. */
21 #include "polybench.h"
22
23 /* Include benchmark-specific header. */
24 #include "adi.h"
25
26 /* Array initialization. */

```

```

27 static void init_array(int n, DATA_TYPE POLYBENCH_2D(u, N, N, n, n)) {
28     int i, j;
29
30     for (i = 0; i < n; i++)
31         for (j = 0; j < n; j++) {
32             u[i][j] = (DATA_TYPE)(i + n - j) / n;
33         }
34 }
35
36 /* DCE code. Must scan the entire live-out data.
37    Can be used also to check the correctness of the output. */
38 static void print_array(int n, DATA_TYPE POLYBENCH_2D(u, N, N, n, n))
39 {
40     {
41         int i, j;
42
43         POLYBENCH_DUMP_START;
44         POLYBENCH_DUMP_BEGIN("u");
45         for (i = 0; i < n; i++)
46             for (j = 0; j < n; j++) {
47                 if ((i * n + j) % 20 == 0)
48                     fprintf(POLYBENCH_DUMP_TARGET, "\n");
49                 fprintf(POLYBENCH_DUMP_TARGET, DATA_PRINTF_MODIFIER, u[i][j]);
50             }
51         POLYBENCH_DUMP_END("u");
52         POLYBENCH_DUMP_FINISH;
53     }
54
55     /* Main computational kernel. The whole function will be timed,
56        including the call and return. */
57     /* Based on a Fortran code fragment from Figure 5 of
58        * "Automatic Data and Computation Decomposition on Distributed Memory Parallel
59        * Computers" by Peizong Lee and Zvi Meir Kedem, TOPLAS, 2002
60        */
61     #pragma clava kernel
62     #pragma clava data kernel :[{
63         scalar : "auto" }, {scalar : "auto" }, {auto : "sizeof(double[n][n])" }, \
64         {auto : "sizeof(double[n][n])" }, {auto : "sizeof(double[n][n])" }, \
65         {auto : "sizeof(double[n][n])" }]
66 void kernel_adi(int tsteps, int n, DATA_TYPE POLYBENCH_2D(u, N, N, n, n),
67                 DATA_TYPE POLYBENCH_2D(v, N, N, n, n),
68                 DATA_TYPE POLYBENCH_2D(p, N, N, n, n),
69                 DATA_TYPE POLYBENCH_2D(q, N, N, n, n)) {
70     int t, i, j;
71     DATA_TYPE DX, DY, DT;
72     DATA_TYPE B1, B2;
73     DATA_TYPE mul1, mul2;
74     DATA_TYPE a, b, c, d, e, f;
75

```

```

76     DX = SCALAR_VAL(1.0) / (DATA_TYPE)_PB_N;
77     DY = SCALAR_VAL(1.0) / (DATA_TYPE)_PB_N;
78     DT = SCALAR_VAL(1.0) / (DATA_TYPE)_PB_TSTEPS;
79     B1 = SCALAR_VAL(2.0);
80     B2 = SCALAR_VAL(1.0);
81     mul1 = B1 * DT / (DX * DX);
82     mul2 = B2 * DT / (DY * DY);
83
84     a = -mul1 / SCALAR_VAL(2.0);
85     b = SCALAR_VAL(1.0) + mul1;
86     c = a;
87     d = -mul2 / SCALAR_VAL(2.0);
88     e = SCALAR_VAL(1.0) + mul2;
89     f = d;
90
91     for (t = 1; t <= _PB_TSTEPS; t++) {
92         for (i = 1; i < _PB_N - 1; i++) {
93             v[0][i] = SCALAR_VAL(1.0);
94             p[i][0] = SCALAR_VAL(0.0);
95             q[i][0] = v[0][i];
96             for (j = 1; j < _PB_N - 1; j++) {
97                 p[i][j] = -c / (a * p[i][j - 1] + b);
98                 q[i][j] = (-d * u[j][i - 1] +
99                     (SCALAR_VAL(1.0) + SCALAR_VAL(2.0) * d) * u[j][i] -
100                     f * u[j][i + 1] - a * q[i][j - 1]) /
101                     (a * p[i][j - 1] + b);
102             }
103
104             v[_PB_N - 1][i] = SCALAR_VAL(1.0);
105             for (j = _PB_N - 2; j >= 1; j--) {
106                 v[j][i] = p[i][j] * v[j + 1][i] + q[i][j];
107             }
108         }
109
110         for (i = 1; i < _PB_N - 1; i++) {
111             u[i][0] = SCALAR_VAL(1.0);
112             p[i][0] = SCALAR_VAL(0.0);
113             q[i][0] = u[i][0];
114             for (j = 1; j < _PB_N - 1; j++) {
115                 p[i][j] = -f / (d * p[i][j - 1] + e);
116                 q[i][j] = (-a * v[i - 1][j] +
117                     (SCALAR_VAL(1.0) + SCALAR_VAL(2.0) * a) * v[i][j] -
118                     c * v[i + 1][j] - d * q[i][j - 1]) /
119                     (d * p[i][j - 1] + e);
120             }
121             u[i][_PB_N - 1] = SCALAR_VAL(1.0);
122             for (j = _PB_N - 2; j >= 1; j--) {
123                 u[i][j] = p[i][j] * u[i][j + 1] + q[i][j];
124             }

```

```

125     }
126 }
127 }
128
129 int main(int argc, char **argv) {
130     /* Retrieve problem size. */
131     int n = N;
132     // int tsteps = TSTEPS;
133
134     /* Variable declaration/allocation. */
135     POLYBENCH_2D_ARRAY_DECL(u, DATA_TYPE, N, N, n, n);
136     POLYBENCH_2D_ARRAY_DECL(v, DATA_TYPE, N, N, n, n);
137     POLYBENCH_2D_ARRAY_DECL(p, DATA_TYPE, N, N, n, n);
138     POLYBENCH_2D_ARRAY_DECL(q, DATA_TYPE, N, N, n, n);
139
140     /* Initialize array(s). */
141     init_array(n, POLYBENCH_ARRAY(u));
142
143     /* Start timer. */
144     polybench_start_instruments;
145
146     /* Run kernel. */
147     for (n = 1; n < N; n++) {
148         kernel_adi(n / 2, n, POLYBENCH_ARRAY(u), POLYBENCH_ARRAY(v),
149                 POLYBENCH_ARRAY(p), POLYBENCH_ARRAY(q));
150     }
151
152     /* Stop and print timer. */
153     polybench_stop_instruments;
154     polybench_print_instruments;
155
156     /* Prevent dead-code elimination. All live-out data must be printed
157        by the function call in argument. */
158     polybench_prevent_dce(print_array(n, POLYBENCH_ARRAY(u)));
159
160     /* Be clean. */
161     POLYBENCH_FREE_ARRAY(u);
162     POLYBENCH_FREE_ARRAY(v);
163     POLYBENCH_FREE_ARRAY(p);
164     POLYBENCH_FREE_ARRAY(q);
165
166     return 0;
167 }

```

Listing B.3: Original CxxSource/adi.cpp

B.1.3 CxxSource/adi.h

```

1  /**
2   * This version is stamped on May 10, 2016
3   *
4   * Contact:
5   *   Louis-Noel Pouchet <pouchet.ohio-state.edu>
6   *   Tomofumi Yuki <tomofumi.yuki.fr>
7   *
8   * Web address: http://polybench.sourceforge.net
9   */
10 #ifndef _ADI_H
11 # define _ADI_H
12
13 /* Default to LARGE_DATASET. */
14 # if !defined(MINI_DATASET) && !defined(SMALL_DATASET) && !defined(MEDIUM_DATASET)
15     && !defined(LARGE_DATASET) && !defined(EXTRALARGE_DATASET)
16 #   define LARGE_DATASET
17 # endif
18
19 # if !defined(TSTEPS) && !defined(N)
20 /* Define sample dataset sizes. */
21 #   ifdef MINI_DATASET
22 #     define TSTEPS 20
23 #     define N 20
24 #   endif
25
26 #   ifdef SMALL_DATASET
27 #     define TSTEPS 40
28 #     define N 60
29 #   endif
30
31 #   ifdef MEDIUM_DATASET
32 #     define TSTEPS 100
33 #     define N 200
34 #   endif
35
36 #   ifdef LARGE_DATASET
37 #     define TSTEPS 500
38 #     define N 1000
39 #   endif
40
41 #   ifdef EXTRALARGE_DATASET
42 #     define TSTEPS 1000
43 #     define N 2000
44 #   endif
45
46 #endif /* !(TSTEPS N) */
47
48 # define _PB_TSTEPS POLYBENCH_LOOP_BOUND(TSTEPS,tsteps)

```

```

49 # define _PB_N POLYBENCH_LOOP_BOUND(N,n)
50
51
52 /* Default data type */
53 # if !defined(DATA_TYPE_IS_INT) && !defined(DATA_TYPE_IS_FLOAT) && !defined(
    DATA_TYPE_IS_DOUBLE)
54 # define DATA_TYPE_IS_DOUBLE
55 # endif
56
57 #ifndef DATA_TYPE_IS_INT
58 # define DATA_TYPE int
59 # define DATA_PRINTF_MODIFIER "%d "
60 #endif
61
62 #ifndef DATA_TYPE_IS_FLOAT
63 # define DATA_TYPE float
64 # define DATA_PRINTF_MODIFIER "%0.2f "
65 # define SCALAR_VAL(x) x##f
66 # define Sqrt_FUN(x) sqrtf(x)
67 # define EXP_FUN(x) expf(x)
68 # define POW_FUN(x,y) powf(x,y)
69 # endif
70
71 #ifndef DATA_TYPE_IS_DOUBLE
72 # define DATA_TYPE double
73 # define DATA_PRINTF_MODIFIER "%0.2lf "
74 # define SCALAR_VAL(x) x
75 # define Sqrt_FUN(x) sqrt(x)
76 # define EXP_FUN(x) exp(x)
77 # define POW_FUN(x,y) pow(x,y)
78 # endif
79
80 #endif /* !_ADI_H */

```

Listing B.4: Original CxxSource/adi.h

B.2 Tribble Output

```

1 | -woven_code
2 | | -_HLS_graphs
3 | | -_HLS_reports
4 | | -CxxIgnored
5 | | | -OCL_Helpers.hpp
6 | | | -OCL_Helpers.cpp
7 | | -CxxSource
8 | | | -adi.cpp

```

```

9 | | |-adi.h
10 | | |-kernels.cpp
11 | | |-polybench.c
12 | | |-polybench.h
13 | | |-CxxTemplates
14 | | |-OCLH_main.cpp

```

Listing B.5: Output Folder Structure

B.2.1 woven_code/CxxTemplates/OCLH_main.cpp

```

1 #include "../CxxIgnored/OCL_Helpers.hpp"
2 cl::Context context;
3 cl::CommandQueue queue;
4 cl::Program program;
5 void scheduler(uint64_t ops[], bool &executeOnCPU, bool &executeOnFPGA,
6               bool &lowConfidence) {
7     executeOnCPU = true;
8     executeOnFPGA = true;
9     lowConfidence = true;
10 }
11
12 void train(uint ops[], bool CPUwonFPGAlost) { return; }
13
14 #pragma clava ocl_insert_globals
15
16 extern int main_original(int argc, char **argv);
17 cl::Kernel krnl_kernel_adi;
18 // cl::Kernel krnl_scheduler;
19 int main(int argc, char *argv[]) {
20     static std::string const platformName = "Xilinx";
21     OCLH::getConfig(argv[1], platformName, context, queue, program);
22 #pragma clava ocl_insert_kernel_initializations
23     krnl_kernel_adi = OCLH::getKernel("kernel_adi_Kernel", program);
24     // krnl_scheduler = OCLH::getKernel("scheduler", program);
25
26     return main_original(argc, argv);
27 }

```

Listing B.6: Generated woven_code/CxxTemplates/OCLH_main.cpp

B.2.2 woven_code/CxxSource/adi.cpp

```

1 #include "adi.h"

```

```

2  #include "../CxxIgnored/OCL_Helpers.hpp"
3  #include "polybench.h"
4  #include <chrono>
5  #include <math.h>
6  #include <stdio.h>
7  #include <string.h>
8  #include <unistd.h>
9  extern void scheduler(uint64_t ops[], bool &executeOnCPU, bool &executeOnFPGA,
10                      bool &lowConfidence);
11 extern void train(uint64_t ops[], bool CPUwonFPGAlost);
12 extern cl::Kernel krnl_kernel_adi;
13 extern cl::CommandQueue queue;
14 extern cl::Context context;
15 /**
16  * This version is stamped on May 10, 2016
17  *
18  * Contact:
19  *   Louis-Noel Pouchet <pouchet.ohio-state.edu>
20  *   Tomofumi Yuki <tomofumi.yuki.fr>
21  *
22  * Web address: http://polybench.sourceforge.net
23  */
24 /*adi.c: this file is part of PolyBench/C*/
25 /*Include polybench common header.*/
26 /*Include benchmark-specific header.*/
27 /*Array initialization.*/
28 static void init_array(int n, float u[2000][2000]) {
29     int i, j;
30     for (i = 0; i < n; i++)
31         for (j = 0; j < n; j++) {
32             u[i][j] = (float)(i + n - j) / n;
33         }
34 }
35
36 /*DCE code. Must scan the entire live-out data.
37 Can be used also to check the correctness of the output.*/
38 static void print_array(int n, float u[2000][2000]) {
39     int i, j;
40     fprintf(stderr, "===BEGIN DUMP_ARRAYS==\n");
41     fprintf(stderr, "begin dump: %s", "u");
42     for (i = 0; i < n; i++)
43         for (j = 0; j < n; j++) {
44             if ((i * n + j) % 20 == 0)
45                 fprintf(stderr, "\n");
46             fprintf(stderr, "%0.2f ", u[i][j]);
47         }
48     fprintf(stderr, "\nend   dump: %s\n", "u");
49     fprintf(stderr, "===END   DUMP_ARRAYS==\n");
50 }

```



```

51
52 void kernel_adi_KernelCount(int tsteps, int n, float u[2000][2000],
53                             float v[2000][2000], float p[2000][2000],
54                             float q[2000][2000], unsigned long ops[]) {
55     ops[0] =
56         (((5 + 1 + 1) * ((n - 1) - 1) + (1 + 1) * ((n - 2) - 1 + 1) + 3 + 1) *
57          ((n - 1) - 1) +
58          ((5 + 1 + 1) * ((n - 1) - 1) + (1 + 1) * ((n - 2) - 1 + 1) + 3 + 1) *
59          ((n - 1) - 1) +
60          1) *
61         ((tsteps)-1 + 1)) *
62         (1);
63     ops[10] = (((2) * ((n - 1) - 1)) * ((n - 1) - 1) +
64               ((2) * ((n - 1) - 1)) * ((n - 1) - 1)) *
65               ((tsteps)-1 + 1) +
66               7) *
67               (1);
68     ops[8] =
69         (((4 + 2) * ((n - 1) - 1) + (1) * ((n - 2) - 1 + 1)) * ((n - 1) - 1) +
70          ((4 + 2) * ((n - 1) - 1) + (1) * ((n - 2) - 1 + 1)) * ((n - 1) - 1)) *
71          ((tsteps)-1 + 1) +
72          2) *
73          (1);
74     ops[9] =
75         (((7) * ((n - 1) - 1) + (1) * ((n - 2) - 1 + 1)) * ((n - 1) - 1) +
76          ((7) * ((n - 1) - 1) + (1) * ((n - 2) - 1 + 1)) * ((n - 1) - 1)) *
77          ((tsteps)-1 + 1) +
78          4) *
79          (1);
80 }
81
82 /*Main computational kernel. The whole function will be timed,
83 including the call and return.*/
84
85 /*Based on a Fortran code fragment from Figure 5 of
86  * "Automatic Data and Computation Decomposition on Distributed Memory Parallel
87  * Computers" by Peizong Lee and Zvi Meir Kedem, TOPLAS, 2002
88  */
89
90 #pragma clava kernel
91
92 #pragma clava data kernel : [{
93     scalar : "auto" }, {scalar : "auto" }, {auto : "sizeof(double[n][n])" }, \
94     {auto : "sizeof(double[n][n])" }, \
95     {auto : "sizeof(double[n][n])" }]
96
97 void kernel_adi(int tsteps, int n, float u[2000][2000], float v[2000][2000],
98                float p[2000][2000], float q[2000][2000]) {
99     bool executeOnCPU;

```

```

100     bool executeOnFPGA;
101     bool measurePerf;
102     std::chrono::high_resolution_clock::duration clava_timing_duration_0;
103     std::chrono::high_resolution_clock::duration clava_timing_duration_1;
104     uint64_t opsCount[16];
105     kernel_adi_KernelCount(tsteps, n, u, v, p, q, opsCount);
106     scheduler(opsCount, executeOnCPU, executeOnFPGA, measurePerf);
107     if (executeOnCPU) {
108         std::chrono::high_resolution_clock::time_point clava_timing_start_0;
109         std::chrono::high_resolution_clock::time_point clava_timing_end_0;
110         if (measurePerf) {
111             clava_timing_start_0 = std::chrono::high_resolution_clock::now();
112         }
113         kernel_adi_Kernel(tsteps, n, u, v, p, q);
114         if (measurePerf) {
115             clava_timing_end_0 = std::chrono::high_resolution_clock::now();
116             clava_timing_duration_0 = clava_timing_end_0 - clava_timing_start_0;
117         }
118     }
119     if (executeOnFPGA) {
120         std::chrono::high_resolution_clock::time_point clava_timing_start_1;
121         std::chrono::high_resolution_clock::time_point clava_timing_end_1;
122         if (measurePerf) {
123             clava_timing_start_1 = std::chrono::high_resolution_clock::now();
124         }
125         krnl_kernel_adi.setArg(0, tsteps);
126         krnl_kernel_adi.setArg(1, n);
127         cl::Buffer buffer_u(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_WRITE,
128                               sizeof(double[n][n]), u);
129         krnl_kernel_adi.setArg(2, buffer_u);
130         cl::Buffer buffer_v(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_WRITE,
131                               sizeof(double[n][n]), v);
132         krnl_kernel_adi.setArg(3, buffer_v);
133         cl::Buffer buffer_p(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_WRITE,
134                               sizeof(double[n][n]), p);
135         krnl_kernel_adi.setArg(4, buffer_p);
136         cl::Buffer buffer_q(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_WRITE,
137                               sizeof(float[2000][2000]), q);
138         krnl_kernel_adi.setArg(5, buffer_q);
139         queue.enqueueMigrateMemObjects({buffer_u, buffer_v, buffer_p, buffer_q},
140                                       0 /* 0 means from host */);
141         queue.enqueueTask(krnl_kernel_adi);
142         queue.finish();
143         queue.enqueueMigrateMemObjects({buffer_u, buffer_v, buffer_p, buffer_q},
144                                       CL_MIGRATE_MEM_OBJECT_HOST);
145         queue.finish();
146         if (measurePerf) {
147             clava_timing_end_1 = std::chrono::high_resolution_clock::now();
148             clava_timing_duration_1 = clava_timing_end_1 - clava_timing_start_1;

```

```

149     }
150 }
151 if (measurePerf)
152     train(opsCount, clava_timing_duration_0 < clava_timing_duration_1);
153 }
154
155 int main_original(int argc, char **argv) {
156     /*Retrieve problem size.*/
157     int n = 2000;
158     // int tsteps = TSTEPS;
159     /*Variable declaration/allocation.*/
160     float(*u)[2000][2000];
161     u = (float(*)[2000][2000])polybench_alloc_data((2000 + 0) * (2000 + 0),
162                                                    sizeof(float));
163     ;
164     float(*v)[2000][2000];
165     v = (float(*)[2000][2000])polybench_alloc_data((2000 + 0) * (2000 + 0),
166                                                    sizeof(float));
167     ;
168     float(*p)[2000][2000];
169     p = (float(*)[2000][2000])polybench_alloc_data((2000 + 0) * (2000 + 0),
170                                                    sizeof(float));
171     ;
172     float(*q)[2000][2000];
173     q = (float(*)[2000][2000])polybench_alloc_data((2000 + 0) * (2000 + 0),
174                                                    sizeof(float));
175     ;
176     /*Initialize array(s).*/
177     init_array(n, *u);
178     /*Start timer.*/
179     ;
180     /*Run kernel.*/
181     for (n = 1; n < 2000; n++) {
182         kernel_adi(n / 2, n, *u, *v, *p, *q);
183     }
184     /*Stop and print timer.*/
185     ;
186     ;
187     /*Prevent dead-code elimination. All live-out data must be printed
188     by the function call in argument.*/
189     if (argc > 42 && !strcmp(argv[0], ""))
190         print_array(n, *u);
191     /*Be clean.*/
192     free((void *)u);
193     ;
194     free((void *)v);
195     ;
196     free((void *)p);
197     ;

```

```

198     free((void *)q);
199     ;
200
201     return 0;
202 }

```

Listing B.7: Generated woven_code/CxxSource/adi.cpp

B.2.3 woven_code/CxxSource/adi.h

```

1 #ifndef _ADI_H_
2 #define _ADI_H_
3
4 #endif

```

Listing B.8: Generated woven_code/CxxSource/adi.h

B.2.4 woven_code/CxxSource/kernels.cpp

```

1 #include "adi.h"
2 #include "polybench.h"
3 #include <math.h>
4 #include <stdio.h>
5 #include <string.h>
6 #include <unistd.h>
7 void kernel_adi_KernelCode(int tsteps, int n, float u[2000][2000],
8                             float v[2000][2000], float p[2000][2000],
9                             float q[2000][2000]) {
10 #pragma HLS array_partition variable = u cyclic factor = 64
11 #pragma HLS array_partition variable = v cyclic factor = 64
12 #pragma HLS array_partition variable = p cyclic factor = 64
13 #pragma HLS array_partition variable = q cyclic factor = 64
14     float f;
15     float e;
16     float d;
17     float c;
18     float b;
19     float a;
20     float mul2;
21     float mul1;
22     float B2;
23     float B1;
24     float DT;
25     float DY;

```

```

26     float DX;
27     int j;
28     int i;
29     int t;
30     DX = 1.0f / (float)n;
31     DY = 1.0f / (float)n;
32     DT = 1.0f / (float)tsteps;
33     B1 = 2.0f;
34     B2 = 1.0f;
35     mul1 = B1 * DT / (DX * DX);
36     mul2 = B2 * DT / (DY * DY);
37     a = -mul1 / 2.0f;
38     b = 1.0f + mul1;
39     c = a;
40     d = -mul2 / 2.0f;
41     e = 1.0f + mul2;
42     f = d;
43     for (t = 1; t <= tsteps; t++) {
44         for (i = 1; i < n - 1; i++) {
45 #pragma HLS pipeline
46             v[0][i] = 1.0f;
47             p[i][0] = 0.0f;
48             q[i][0] = v[0][i];
49             for (j = 1; j < n - 1; j++) {
50 #pragma HLS unroll
51                 p[i][j] = -c / (a * p[i][j - 1] + b);
52                 q[i][j] = (-d * u[j][i - 1] + (1.0f + 2.0f * d) * u[j][i] -
53                     f * u[j][i + 1] - a * q[i][j - 1]) /
54                     (a * p[i][j - 1] + b);
55             }
56             v[n - 1][i] = 1.0f;
57             for (j = n - 2; j >= 1; j--) {
58 #pragma HLS unroll
59                 v[j][i] = p[i][j] * v[j + 1][i] + q[i][j];
60             }
61         }
62         for (i = 1; i < n - 1; i++) {
63 #pragma HLS pipeline
64             u[i][0] = 1.0f;
65             p[i][0] = 0.0f;
66             q[i][0] = u[i][0];
67             for (j = 1; j < n - 1; j++) {
68 #pragma HLS unroll
69                 p[i][j] = -f / (d * p[i][j - 1] + e);
70                 q[i][j] = (-a * v[i - 1][j] + (1.0f + 2.0f * a) * v[i][j] -
71                     c * v[i + 1][j] - d * q[i][j - 1]) /
72                     (d * p[i][j - 1] + e);
73             }
74             u[i][n - 1] = 1.0f;

```

```
75         for (j = n - 2; j >= 1; j--) {
76 #pragma HLS unroll
77             u[i][j] = p[i][j] * u[i][j + 1] + q[i][j];
78         }
79     }
80 }
81 }
82
83 void kernel_adi_Kernel(int tsteps, int n, float u[2000][2000],
84                       float v[2000][2000], float p[2000][2000],
85                       float q[2000][2000]) {
86     kernel_adi_KernelCode(tsteps, n, u, v, p, q);
87 }
```

Listing B.9: Generated woven_code/CxxSource/kernels.cpp

Appendix C

adi Kernel Optimisations

Note: A reduction of the max matrix size from 2000 to 400 for each dimension was necessary to produce a synthesisable kernel.

```
1  /* #include "adi.h"
2  #include "polybench.h"
3  #include <math.h>
4  #include <stdio.h>
5  #include <string.h>
6  #include <unistd.h> */
7
8  extern "C" {
9  void kernel_adi_KernelCode(int tsteps, int n, float u_o[400][400],
10                             float v_o[400][400], float p_o[400][400],
11                             float q_o[400][400]) {
12      float u[400][400];
13      float v[400][400];
14      float p[400][400];
15      float q[400][400];
16
17  u_read_out:
18      for (int i = 0; i < n; i++) {
19          u_read_int:
20              for (int j = 0; j < n; j++) {
21                  u[i][j] = u_o[i][j];
22              }
23      }
24  v_read_out:
25      for (int i = 0; i < n; i++) {
26          v_read_int:
27              for (int j = 0; j < n; j++) {
28                  v[i][j] = v_o[i][j];
29              }
30      }
31  }
```



```

81         v[n - 1][i] = 1.0f;
82         for (j = n - 2; j >= 1; j--) {
83 #pragma HLS unroll
84             v[j][i] = p[i][j] * v[j + 1][i] + q[i][j];
85         }
86     }
87     for (i = 1; i < n - 1; i++) {
88 #pragma HLS pipeline
89         u[i][0] = 1.0f;
90         p[i][0] = 0.0f;
91         q[i][0] = u[i][0];
92         for (j = 1; j < n - 1; j++) {
93 #pragma HLS unroll
94             p[i][j] = -f / (d * p[i][j - 1] + e);
95             q[i][j] = (-a * v[i - 1][j] + (1.0f + 2.0f * a) * v[i][j] -
96                 c * v[i + 1][j] - d * q[i][j - 1]) /
97                 (d * p[i][j - 1] + e);
98         }
99         u[i][n - 1] = 1.0f;
100        for (j = n - 2; j >= 1; j--) {
101 #pragma HLS unroll
102            u[i][j] = p[i][j] * u[i][j + 1] + q[i][j];
103        }
104    }
105 }
106
107 p_read_out:
108     for (int i = 0; i < n; i++) {
109         p_read_int:
110             for (int j = 0; j < n; j++) {
111                 p_o[i][j] = p[i][j];
112             }
113     }
114 q_read_out:
115     for (int i = 0; i < n; i++) {
116         q_read_int:
117             for (int j = 0; j < n; j++) {
118                 q_o[i][j] = q[i][j];
119             }
120     }
121 }
122
123 void kernel_adi_float_Kernel(int tsteps, int n, float u[400][400],
124                             float v[400][400], float p[400][400],
125                             float q[400][400]) {
126     kernel_adi_KernelCode(tsteps, n, u, v, p, q);
127 }
128 }

```

Listing C.1: Slightly modified *adi* kernel code. Optimised for FPGA synthesis

Appendix D

Dataset Creation Code

```
1  #include <fstream>
2  #include <iostream>
3  #include <sstream>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string>
7
8  #include "src/Tree.hpp"
9
10 #define WRITE_TO_FILE
11
12 const std::string outFilename = "dataset.txt";
13
14 void kernel_adi_KernelCount(uint64_t tsteps, uint64_t n, uint64_t ops[]) {
15     ops[0] =
16         (((5 + 1 + 1) * ((n - 1) - 1) + (1 + 1) * ((n - 2) - 1 + 1) + 3 + 1) *
17         ((n - 1) - 1) +
18         ((5 + 1 + 1) * ((n - 1) - 1) + (1 + 1) * ((n - 2) - 1 + 1) + 3 + 1) *
19         ((n - 1) - 1) +
20         1) *
21         ((tsteps)-1 + 1)) *
22         (1);
23     ops[10] = (((2) * ((n - 1) - 1)) * ((n - 1) - 1) +
24         ((2) * ((n - 1) - 1)) * ((n - 1) - 1)) *
25         ((tsteps)-1 + 1) +
26         7) *
27         (1);
28     ops[8] =
29         (((4 + 2) * ((n - 1) - 1) + (1) * ((n - 2) - 1 + 1)) * ((n - 1) - 1) +
30         ((4 + 2) * ((n - 1) - 1) + (1) * ((n - 2) - 1 + 1)) * ((n - 1) - 1)) *
31         ((tsteps)-1 + 1) +
32         2) *
33         (1);
```

```

34     ops[9] =
35         (((7) * ((n - 1) - 1) + (1) * ((n - 2) - 1 + 1)) * ((n - 1) - 1) +
36         ((7) * ((n - 1) - 1) + (1) * ((n - 2) - 1 + 1)) * ((n - 1) - 1)) *
37         ((tsteps)-1 + 1) +
38         4) *
39         (1);
40 }
41
42 void kernel_2mm_KernelCount(uint64_t ni, uint64_t nj, uint64_t nk, uint64_t nl,
43                             uint64_t ops[]) {
44     ops[0] = (((1 * nk + 1) * nj + 1) * ni + ((1 * nj + 1) * nl + 1) * ni) * 1;
45     ops[12] = (((1 * nk) * nj) * ni + ((1 * nj) * nl) * ni) * 1;
46     ops[13] = (((2 * nk) * nj) * ni + ((1 * nj + 1) * nl) * ni) * 1;
47 }
48
49 void kernel_3mm_KernelCount(uint64_t ni, uint64_t nj, uint64_t nk, uint64_t nl,
50                             uint64_t nm, uint64_t ops[]) {
51     ops[0] = (((1 * nk + 1) * nj + 1) * ni + ((1 * nm + 1) * nl + 1) * nj +
52             ((1 * nj + 1) * nl + 1) * ni) *
53             1;
54     ops[12] =
55         (((1 * nk) * nj) * ni + ((1 * nm) * nl) * nj + ((1 * nj) * nl) * ni) *
56         1;
57     ops[13] =
58         (((1 * nk) * nj) * ni + ((1 * nm) * nl) * nj + ((1 * nj) * nl) * ni) *
59         1;
60 }
61
62 void kernel_atax_KernelCount(uint64_t m, uint64_t n, uint64_t ops[]) {
63     ops[0] = (1 * n + (1 * n + 1 * n + 1) * m) * 1;
64     ops[12] = ((1 * n + 1 * n) * m) * 1;
65     ops[13] = ((1 * n + 1 * n) * m) * 1;
66 }
67
68 void kernel_bicg_KernelCount(uint64_t m, uint64_t n, uint64_t ops[]) {
69     ops[0] = (1 * m + (1 * m + 1) * n) * 1;
70     ops[12] = ((2 * m) * n) * 1;
71     ops[13] = ((2 * m) * n) * 1;
72 }
73
74 void kernel_deriche_KernelCount(uint64_t w, uint64_t h, uint64_t ops[]) {
75     ops[0] =
76         ((1 * h + 1) * w + (1 * ((h - 1) + 1) + 1 + 1) * w + (1 * h + 1) * w +
77         (1 * w + 1) * h + (1 * ((w - 1) + 1) + 1 + 1) * h + (1 * h + 1) * w) *
78         1;
79     ops[8] = (((3) * ((h))) * ((w)) + ((3) * ((h - 1) + 1)) * ((w)) +
80             ((1) * ((h))) * ((w)) + ((3) * ((w))) * ((h)) +
81             ((3) * ((w - 1) + 1)) * ((h)) + ((1) * ((h))) * ((w)) + 4 + 2) *
82             (1);

```

```

83     ops[9] = (((4) * ((h))) * ((w)) + ((4) * ((h - 1) + 1)) * ((w)) +
84              ((1) * ((h))) * ((w)) + ((4) * ((w))) * ((h)) +
85              ((4) * ((w - 1) + 1)) * ((h)) + ((1) * ((h))) * ((w)) + 11) *
86              (1);
87     ops[10] = 1 * 1;
88 }
89
90 int main() {
91
92     std::ofstream fout(outFilename);
93     float split = 0;
94
95     //////////
96     // adi //
97     //////////
98 #define MIN 20
99 #define MAX 2000
100     for (uint n = MIN; n <= MAX; n++) {
101
102         uint64_t ops[Tree::_DataClass::N_Attributes] = {0};
103         kernel_adi_KernelCount(n, n / 2, ops);
104         bool fpgaWonCPULost = n > ((MAX - MIN) / 2);
105
106 #ifdef WRITE_TO_FILE
107         for (uint i = 0; i < Tree::_DataClass::N_Attributes; i++) {
108             if (i) {
109                 fout << " ";
110             }
111             fout << std::to_string(ops[i]);
112         }
113         fout << " " << fpgaWonCPULost << std::endl;
114 #endif
115
116         split += fpgaWonCPULost;
117     }
118
119     std::cout << "ADI split: " << split / (MAX - MIN) << std::endl;
120     split = 0;
121
122     //////////
123     // 2mm //
124     //////////
125 #undef MIN
126 #undef MAX
127 #define MIN 16
128 #define MAX 2200
129     for (uint n = MIN; n <= MAX; n++) {
130
131         uint64_t ops[Tree::_DataClass::N_Attributes] = {0};

```

```

132     kernel_2mm_KernelCount(n, n, n, n, ops);
133     bool fpgaWonCPULost = n > ((MAX - MIN) / 2);
134
135 #ifdef WRITE_TO_FILE
136     for (uint i = 0; i < Tree::_DataClass::N_Attributes; i++) {
137         if (i) {
138             fout << " ";
139         }
140         fout << std::to_string(ops[i]);
141     }
142     fout << " " << fpgaWonCPULost << std::endl;
143 #endif
144
145     split += fpgaWonCPULost;
146 }
147
148     std::cout << "2MM split: " << split / (MAX - MIN) << std::endl;
149     split = 0;
150
151     ////////////
152     // atax //
153     ////////////
154 #undef MIN
155 #undef MAX
156 #define MIN 32
157 #define MAX 2200
158     for (uint n = MIN; n <= MAX; n++) {
159
160         uint64_t ops[Tree::_DataClass::N_Attributes] = {0};
161         kernel_atax_KernelCount(n, n, ops);
162         bool fpgaWonCPULost = n > ((MAX - MIN) / 2);
163
164 #ifdef WRITE_TO_FILE
165         for (uint i = 0; i < Tree::_DataClass::N_Attributes; i++) {
166             if (i) {
167                 fout << " ";
168             }
169             fout << std::to_string(ops[i]);
170         }
171         fout << " " << fpgaWonCPULost << std::endl;
172 #endif
173
174         split += fpgaWonCPULost;
175     }
176
177     std::cout << "ATAX split: " << split / (MAX - MIN) << std::endl;
178     split = 0;
179
180     ////////////

```

```

181     // bicg //
182     //////////
183 #undef MIN
184 #undef MAX
185 #define MIN 32
186 #define MAX 2200
187     for (uint n = MIN; n <= MAX; n++) {
188
189         uint64_t ops[Tree::_DataClass::N_Attributes] = {0};
190         kernel_bicg_KernelCount(n, n, ops);
191         bool fpgaWonCPULost = n > ((MAX - MIN) / 2);
192
193 #ifdef WRITE_TO_FILE
194         for (uint i = 0; i < Tree::_DataClass::N_Attributes; i++) {
195             if (i) {
196                 fout << " ";
197             }
198             fout << std::to_string(ops[i]);
199         }
200         fout << " " << fpgaWonCPULost << std::endl;
201 #endif
202
203         split += fpgaWonCPULost;
204     }
205
206     std::cout << "BICG split: " << split / (MAX - MIN) << std::endl;
207     split = 0;
208
209     //////////
210     // deriche //
211     //////////
212 #undef MIN
213 #undef MAX
214 #define MIN 64
215 #define MAX 4320
216     for (uint n = MIN; n <= MAX; n++) {
217
218         uint64_t ops[Tree::_DataClass::N_Attributes] = {0};
219         kernel_deriche_KernelCount(uint(1.5f * n), n, ops);
220         bool fpgaWonCPULost = n > ((MAX - MIN) / 2);
221
222 #ifdef WRITE_TO_FILE
223         for (uint i = 0; i < Tree::_DataClass::N_Attributes; i++) {
224             if (i) {
225                 fout << " ";
226             }
227             fout << std::to_string(ops[i]);
228         }
229         fout << " " << fpgaWonCPULost << std::endl;

```

```
230 #endif
231
232     split += fpgaWonCPULost;
233 }
234
235 std::cout << "DERICHE split: " << split / (MAX - MIN) << std::endl;
236 split = 0;
237
238 fout.close();
239 }
```

Listing D.1: Dataset creation code.

References

- [1] Mark Horowitz. 1.1 Computing’s energy problem (and what we can do about it). In *Digest of Technical Papers - IEEE International Solid-State Circuits Conference*, volume 57, pages 10–14, 2014. doi:10.1109/ISSCC.2014.6757323.
- [2] Abderazak Ben Abdallah. Heterogeneous Computing: An Emerging Paradigm of Embedded Systems Design. In *Computational Frameworks: Systems, Models and Applications*, pages 61–93. Elsevier, 2017. doi:10.1016/B978-1-78548-256-4.50003-X.
- [3] Xilinx Inc. Zynq UltraScale+ MPSoC ZCU102, 2020. URL: <https://www.xilinx.com/products/boards-and-kits/ek-ul-zcu102-g.html>.
- [4] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *FPGA 2015 - 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170, New York, NY, USA, 2 2015. Association for Computing Machinery, Inc. URL: <https://dl.acm.org/doi/10.1145/2684746.2689060>, doi:10.1145/2684746.2689060.
- [5] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 26, pages 203–215, 2 2007. doi:10.1109/TCAD.2006.884574.
- [6] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. Edge Intelligence: Paving the Last Mile of Artificial Intelligence With Edge Computing. *Proceedings of the IEEE*, 2019. doi:10.1109/JPROC.2019.2918951.
- [7] Intel. High-Level Synthesis Compiler, 2020. URL: <https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html>.
- [8] Xilinx Inc. Vivado High-Level Synthesis. Technical report, Online, 2020. URL: <https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html>.
- [9] Xilinx Inc. Dynamic Function eXchange - Vivado Design Suite User Guide. Technical report, Online, 2020. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug909-vivado-partial-reconfiguration.pdf.
- [10] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann, sixth edition, 2019.

- [11] Usman Dastgeer, Lu Li, and Christoph Kessler. Adaptive implementation selection in the SkePU skeleton programming library. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8299 LNCS, pages 170–183. Springer, Berlin, Heidelberg, 2013. URL: https://link.springer.com/chapter/10.1007/978-3-642-45293-2_13, doi:10.1007/978-3-642-45293-2_{_}13.
- [12] Gina Yuan, Shoumik Palkar, Deepak Narayanan, and Matei Zaharia. Offload annotations: Bringing heterogeneous computing to existing libraries and workloads. In *Proceedings of the 2020 USENIX Annual Technical Conference, ATC 2020*, pages 293–306, 2020. URL: <https://www.usenix.org/conference/atc20/presentation/yuan>.
- [13] Gavin Vaz, Heinrich Riebler, Tobias Kenter, and Christian Plessl. Deferring accelerator offloading decisions to application runtime. In *2014 International Conference on ReConfigurable Computing and FPGAs (ReConFig14)*, pages 1–8, Cancun, Mexico, 12 2014. IEEE. URL: <http://ieeexplore.ieee.org/document/7032509/>, doi:10.1109/ReConFig.2014.7032509.
- [14] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization, CGO*, pages 75–86, 2004. doi:10.1109/CGO.2004.1281665.
- [15] William F. Ogilvie, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Fast automatic heuristic construction using active learning. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8967, pages 146–160. Springer Verlag, 2015. URL: https://link.springer.com/chapter/10.1007/978-3-319-17473-0_10, doi:10.1007/978-3-319-17473-0_{_}10.
- [16] D. del Rio Astorga, Manuel F. Dolz, Javier Fernandez, and Javier Garcia Blas. Hybrid static–dynamic selection of implementation alternatives in heterogeneous environments. *Journal of Supercomputing*, 75(8):4098–4113, 8 2019. URL: <https://doi.org/10.1007/s11227-017-2147-y>, doi:10.1007/s11227-017-2147-y.
- [17] J. Ross Quinlan. Learning Efficient Classification Procedures and Their Application to Chess End Games. *Machine Learning*, pages 463–482, 1983. URL: https://link.springer.com/chapter/10.1007/978-3-662-12405-5_15, doi:10.1007/978-3-662-12405-5_{_}15.
- [18] Paul E. Utgoff. ID5: An Incremental ID3. In *Machine Learning Proceedings 1988*, pages 107–120. Elsevier, 1 1988. URL: <https://linkinghub.elsevier.com/retrieve/pii/B9780934613644500177>, doi:10.1016/b978-0-934613-64-4.50017-7.
- [19] Paul E. Utgoff. Improved Training Via Incremental Learning. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 362–365. Elsevier, 1 1989. URL: <https://linkinghub.elsevier.com/retrieve/pii/B9781558600362500928>, doi:10.1016/b978-1-55860-036-2.50092-8.
- [20] Paul E. Utgoff, Neil C. Berkman, and Jeffery A. Clouse. Decision Tree Induction Based on Efficient Tree Restructuring. *Machine Learning 1997* 29:1, 29(1):5–44, 1997. URL: <https://link.springer.com/article/10.1023/A:1007413323501>, doi:10.1023/A:1007413323501.

- [21] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceeding of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 71–80, New York, New York, USA, 2000. Association for Computing Machinery (ACM). URL: <http://portal.acm.org/citation.cfm?doid=347090.347107>, doi:10.1145/347090.347107.
- [22] Geoff Hulten and Pedro Domingos. VFML—a toolkit for mining high-speed time-changing data streams, 2003. URL: <http://www.cs.washington.edu/dm/vfml/>.
- [23] Bernhard Pfahringer, Geoffrey Holmes, and Richard Kirkby. Handling Numeric Attributes in Hoëffding Trees. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5012 LNAI:296–307, 2008. URL: https://link.springer.com/chapter/10.1007/978-3-540-68125-0_27, doi:10.1007/978-3-540-68125-0{_}27.
- [24] Zhe Lin, Sharad Sinha, and Wei Zhang. Towards Efficient and Scalable Acceleration of Online Decision Tree Learning on FPGA. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 172–180. IEEE, 4 2019. URL: <https://ieeexplore.ieee.org/document/8735508>, doi:10.1109/FCCM.2019.00032.
- [25] Alric Althoff and Ryan Kastner. An Architecture for Learning Stream Distributions with Application to RNG Testing. *Proceedings - Design Automation Conference*, Part 128280, 6 2017. URL: <http://dx.doi.org/10.1145/3061639.3062199>, doi:10.1145/3061639.3062199.
- [26] João Bispo and João M.P. Cardoso. Clava: C/C++ source-to-source compilation using LARA. *SoftwareX*, 12:100565, 7 2020. doi:10.1016/J.SOFTX.2020.100565.
- [27] Xilinx Inc. Vitis Unified Software Platform. URL: <https://www.xilinx.com/products/design-tools/vitis.html>.
- [28] Tiago Lascasas dos Santos. *Acceleration of Applications with FPGA-based Computing Machines: Code Restructuring*. PhD thesis, University of Porto, Porto, Portugal, 7 2020. URL: <https://hdl.handle.net/10216/128984>.
- [29] mathieujois. GitHub - mlrequest/sklearn-json: A safe, transparent way to share and deploy scikit-learn models. URL: <https://github.com/mlrequest/sklearn-json>.
- [30] PolyBench/C – Homepage of Louis-Noël Pouchet. URL: <http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/>.
- [31] Nuno Paulino. A Generator of Randomly Correlated N-Dimensional Clusters, 2020. URL: https://www.researchgate.net/publication/343255786_A_Generator_of_Randomly_Correlated_N-Dimensional_Clusters, doi:10.13140/RG.2.2.34866.43200.