http://researchspace.auckland.ac.nz

ResearchSpace@Auckland

Copyright Statement

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

This thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of this thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from their thesis.

To request permissions please use the Feedback form on our webpage.
http://researchspace.auckland.ac.nz/feedback

General copyright and disclaimer

In addition to the above conditions, authors give their consent for the digital copy of their work to be used subject to the conditions specified on the Library Thesis Consent Form and Deposit Licence.

NOVEL APPROACHES TO AUTOMATIC HARDWARE ACCELERATION OF HIGH-LEVEL SOFTWARE

RAVIKESH CHANDRA

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy in Computer Systems Engineering, The University of Auckland, May 2013

Ravikesh Chandra: *Novel approaches to automatic hardware acceleration of high-level software* © May 2013

ABSTRACT

Reconfigurable computing combines traditional processors together with FPGAs, creating heterogeneous architectures ripe for massively improving application performance. Yet, hardware development for FPGAs is notoriously difficult and far-removed from software design, leaving this potential unrealised. This thesis explores two major techniques to address this gap.

The first technique is the seamless integration of dedicated hardware data structures within existing software applications, an area which has received very little attention. Implementing data structures in hardware and exposing them at run-time, can boost the performance of applications.

A case study explored the use of a hardware priority queue in graph algorithms. This implementation attained much better performance characteristics compared to software-only counterparts. Seamless communication between accelerator and the host CPU has been achieved by developing an application abstraction layer with run-time support to choose underlying implementations. This approach increases ease of use given the minimal modifications required to the original application. Moreover, hardware/software co-design is employed to create a hybrid priority queue. This provides tangible benefits, serving as the driver for new features that would be difficult to implement with hardware alone. Complete application experiments showed a moderate overall performance speedup but, more importantly, demonstrated the promise of the concept.

The second technique, the major focus of this thesis, is polyhedral-assisted accelerator generation for loop kernels. Nested loop kernels consisting of numeric operations is a primary, but non-trivial, target for FPGA acceleration. High-level application synthesis addresses

the design challenge by attempting to generate accelerators based on the existing software implementation of the kernel. This thesis extends this concept, using the polyhedral model for the analysis and transformation of the input codes based on a user-specified scattering function. An experimental tool-chain, named polyAcc, was developed which provides a semi-automated implementation of the proposed methodology.

The foundation of this approach is the development of an innovative architectural framework that is amenable to the mapping of accelerator codes. One of the novel proposals is a technique for the exploitation of embedded memories on the FPGA to leverage high bandwidth for computation.

Polyhedral compilation techniques, driven from the behaviour expressed by input scattering functions, form the basis for scheduling and building the accelerator. The thesis investigates methods to generate the datapath, interconnection network, and the accelerator control program from the target polyhedron schedule. Furthermore, scalability and performance are enhanced by applying pipelining and tiling techniques to the designs.

Extensive experimental testing has shown success with different common scientific input kernels. Performance scaled admirably with resource consumption and proved competitive with powerful x86 CPUs.

In loving memory of my grandfathers, who encouraged me to take this journey.

ACKNOWLEDGMENTS

I must sincerely thank Dr Oliver Sinnen, my supervisor, mentor, and friend. He has provided wonderful advice, shown incredible commitment and good faith, and always been there when I have needed him.

Finally, this thesis would not have been possible without the everlasting love of my family. Their unfailing support and patience throughout this journey has been incredible.

CONTENTS

1 INTRODUCTION 1
- 1.1 Field programmable gate array (FPGA) technology 3
- 1.2 Motivation and contributions 5
- 1.3 Publications 7
- 1.4 Thesis structure 7

2 BACKGROUND 9
- 2.1 Reconfigurable computing systems 9
  - 2.1.1 RC system model 11
  - 2.1.2 Computational characteristics 13
- 2.2 Development for reconfigurable systems 15
- 2.3 Hardware design languages 17
  - 2.3.1 Architecture description languages 20
  - 2.3.2 System-level design 20
- 2.4 Application synthesis 22
  - 2.4.1 Low-level application synthesis 23
  - 2.4.2 High-level application synthesis 24
- 2.5 Case study: acceleration using C2H Compiler 25
  - 2.5.1 Complete SoPC example 27
  - 2.5.2 Optimisations and compiler directives 27
  - 2.5.3 Impact of manual optimisation 30
- 2.6 Polyhedral compilation 31
- 2.7 Overview of this thesis 33

3 HARDWARE DATA STRUCTURES 35
- 3.1 Graph processing and the Priority Queue 36
  - 3.1.1 Graph theory 36
  - 3.1.2 Prim's algorithm for computing Minimum Spanning Tree 37
  - 3.1.3 Software priority queue implementation 39
- 3.2 Hardware priority queue 40

3.2.1 Hardware PQ top-level architecture 42
3.2.2 Shift-block storage element architecture 42
3.2.3 Performance characteristics 44
3.3 Hardware/software interface 45
3.3.1 Java/Hardware interface implementation 46
3.3.2 Transparent application usage 47
3.4 Design for scalability 48
3.4.1 Hybrid HW/SW queue for length extension 48
3.4.2 Priority range mapping 50
3.4.3 Extending functionality in the software domain 50
3.5 Performance results 51
3.5.1 Implementation environment 51
3.5.2 Test parameters and synthesis results 52
3.5.3 Performance comparison 53
3.5.4 Embedded vs workstation platform 55
3.6 Conclusions and future directions 56
4 ACCELERATOR HARDWARE MODEL 61
4.1 FPGA physical characteristics 61
4.1.1 Resources 62
4.1.2 Connectivity 63
4.2 Design overview 64
4.3 Datapath architecture 65
4.3.1 Processing element (PE) 65
4.3.2 RAM storage 66
4.3.3 Register storage 68
4.3.4 Interconnection network 69
4.4 System integration 70
4.5 Control mechanisms 72
4.6 Pipelined resource sharing 74
4.6.1 Pipeline model 74
4.6.2 Pipelined PE design 76
4.6.3 Control entry interpretation 76
4.6.4 Communication arbitration 78
5 POLYHEDRAL KERNEL MAPPING 81
5.1 Polyhedral modelling 82
5.1.1 Data access vector equations 83
5.1.2 Data dependences 83
5.2 Scattering functions 85
5.2.1 Time schedule 86

5.2.2 Processor allocation 87
5.2.3 Transformation of a stencil kernel 87
5.3 Multiple target dimensions 89
5.3.1 Lexicographic time ordering 90
5.3.2 Scheduling multiple loop bodies 92
5.3.3 Virtual processor dimensions 93
5.4 Tiling the target polyhedron 96
5.4.1 Tile characteristics 97
5.4.2 Execution strategies for tiling 98
5.4.3 A note on tile shape 99
5.5 Practical considerations 99
5.5.1 Acceptable input 99
5.5.2 Typical transformations 100
6 ACCELERATOR GENERATION 103
6.1 Generation process 103
6.2 Processing element generation 104
6.2.1 Datapath formulation 105
6.2.2 HDL code generation 108
6.3 Storage element generation 111
6.3.1 Storage element analysis 111
6.3.2 Handling multiple dimensions 115
6.3.3 Storage element coherence and reduction 115
6.4 Generation of controller code 116
6.4.1 Accelerator CFG 117
6.4.2 Statement to resource substitutions 119
6.4.3 Nested time dimensions 120
6.4.4 Code generation 121
6.4.5 Virtual processor dimensions 122
6.4.6 Module tiling 123
6.5 Interconnect generation 126
6.5.1 Stencil kernel example 128
7 EXPERIMENTAL EVALUATION 131
7.1 Introduction 131
7.2 Experimental setup 132
7.2.1 Inputs 132
7.2.2 Process 135
7.2.3 Target systems 137
7.2.4 Software OpenMP performance 138
7.3 Non-pipelined accelerators 139

7.3.1 Synthesis results 139
7.3.2 Performance analysis 143
7.3.3 Design-space exploration and usability 146
7.4 Pipelined accelerators 147
7.4.1 Increased accelerator efficiency 147
7.4.2 Targeting large problems 149
7.4.3 Limitations to scalability 150
7.4.4 Performance analysis 151
7.5 Tiled problem space 152
7.5.1 Validation of synthesis characteristics 152
7.5.2 Multi-unit scalability 154
8 CONCLUSIONS 157

BIBLIOGRAPHY 163

1 INTRODUCTION

The general-purpose microprocessor (GPP) is the brain of a typical computer, composed of a balanced set of hardware for the sequential execution of application code. Since they can be easily programmed, they have been used to perform a multitude of tasks within society—spearheading the Information Age and the development and popularisation of the Internet. In the last several decades, the performance of GPPs advanced at a rapid pace, epitomised by Moore's Law[1], resulting in substantial improvements from architectural innovation and raw frequency increases. Thanks to this, the application software gained these performance benefits *for free*.

However, new computing demands have lead to significantly more complex software along with greater abstraction and overhead necessary to manage this complexity. Coupled with the burgeoning demand in new markets, for example, mobile devices, the capabilities of GPPs are being tested; there is an ever omnipresent demand for more processing power and lower power consumption. Although we can continue to expect individual processor performance to increase, it is becoming exponentially more difficult to achieve substantial improvements as these generic architectures have matured and the low-hanging fruit has been picked.

This has culminated with the GPP vendors taking a different approach and duplicating the entire processor core on-die, thus leading to the era of multi-core computing [41]. While parallel systems have existed for decades, the development of multi-core GPPs really

1 Although it is not really a law, Moore predicted the transistor count on microprocessors to double every 24 months allowing for exponentially more powerful and capable computer systems [75].

brought parallel programming into the mainstream. *Theoretically* this approach can provide us an up-to-linear increase in performance as we increase core count! In reality, however, improved performance is not always realised since it depends on the ability of software to explicitly make use of the available parallelism. Unlike the preceding frequency boosts and architectural enhancements of GPPs, this change signal that the era of 'free' performance increases is approaching extinction [97]. Software must now be designed, or re-designed, with parallelism in mind and this is not a trivial task. Parallel software development requires re-training, better tool sets, a strong understanding of the application, and in many areas it is simply not worth pursuing due to the inherent sequential and inter-dependent nature of the computational tasks.

Harnessing the potential of multi-core processors is difficult and comes at a high cost. For this reason there is a computational gap whereby application demands are increasing ever so rapidly but available processing power cannot keep pace.

Recently, heterogeneous computing architectures have become popular. Whereby traditional GPPs are combined with dedicated application or domain specific hardware to accelerate computational workloads. Since accelerators are much less generic than GPPs, they have the potential to perform certain tasks an order of magnitude or faster than the generic cores, while possibly using less power! Heterogenous computing can take different forms and connectedness: it can refer to additional hardware blocks directly embedded within the GPP package to externally connected accelerator hardware over an interconnection network. Communications have primarily been performed over the main system peripheral bus—specifically, some flavour of PCI Express (PCIe).

PCIe is a high-speed, serial, point-to-point computer expansion bus with universal popularity. It is scalable in transfer speeds and data can be striped across 1 to 32 lanes.

More recently, efforts have been made to move accelerator technology onto the same die as the GPP, to improve communication performance, potential memory sharing and coherency, and power consumption. This type of configuration has become coined as accelerated processing units (APUs) [103] and more recently popularised by AMD's hybrid SoPC products. The Cell processor and, more recently, the Xeon Phi can be considered as APU-class products. The Cell [78] follows the more typical design layout of a small GPP with multiple special-purpose acceleration processors. However, the Xeon Phi (derived from [88]) is a many-core chip where each processor has

very wide and capable vector units, coherent L2 cache, and a very wide ring-bus for communication.

Graphics processing units (GPUs) have helped to kickstart and popularise this approach in the mainstream. Architecturally, GPUs are extremely capable for high-performance floating-point calculations (which are necessary for the graphics rendering in contemporary 3D games). GPU vendors have increased the programmability of each underlying processing core, along with the driver stack, such that they can be leveraged for a variety of computational tasks. That said, there are still many challenges faced when using this programming model, along with competing APIs (CUDA and OpenCL) [57], leaving it still a niche domain.

Since GPUs and APUs are fixed-function accelerators they provide a rigid memory and and computational architecture. Developers must be careful to manually design their application to consider these characteristics, in particular around the mapping of data to caches. This is proved to be one of the major challenges for utilising this type of acceleration and, in some respects, is more complex than GPP parallel programming.

1.1 FIELD PROGRAMMABLE GATE ARRAY (FPGA) TECHNOLOGY

FPGAs present another option for accelerating computing that can be utilised to address the computational gap for emerging applications. FGPAs are semiconductor devices that are functionally programmable; they consist of digital components that can be configured in-field to a desired functional hardware specification. This means that they can be configured into dedicated hardware devices exactly as and when required! This flexibility can allow an engineer to configure the FPGA into an *optimised heterogeneous design that specifically targets application requirements* and can realise huge potential performance increases. The parallelism in reconfigurable hardware is also potentially much higher than in software and can be exploited on a much lower level so even greater speed-ups are possible if applications can be formulated in a parallel way. Moreover, the memory architecture is also configurable to allow creative datapath bandwidth optimisation.

Since FPGAs are in competition to GPUs and other APUs, such as the Cell architecture, it is inevitable to make comparative evaluations,

such as in [10, 55, 56, 77]. However, it would be fair to say the conclusion as to what technology is 'better' is that: *it depends*. There are many variables at play and it makes it extremely difficult to make generalised statements on this topic, outside of a realistic application context. Meanwhile, each technology continues to evolve, but FPGAs have the highest potential for further significant improvements for allowing completely custom designs.

Combining the flexible FPGA device into a computing system alongside a traditional GPP will create an extremely powerful and flexible platform which can usher in an era of reconfigurable computing (RC). This is all about the marriage of configurable software-directed processor architectures with configurable hardware architectures. This evolution in computing marks the change from the von Neumann model of computing based on fixed hardware. Although this creates a large opportunity for potential processing power through tailor-made optimised hardware design, to appreciate this new model a paradigm shift in thinking is required [51].

Developing a hardware design for a given application is much more complex, time consuming, and error prone than writing traditional software applications. Different design languages are used for hardware descriptions; the tool chains, build processes, and deployment is very different. In fact, the entire application must be architected in significantly different fashion to a typical micro-processor targeted software application [32]. Ultimately these barriers mean that FPGA devices and reconfigurable computing have been largely ignored and treated as an esoteric domain–*'too difficult, not worth it'*.

BREAKING DOWN THE ADOPTABILITY BARRIERS Making FPGAs more attractive to developers has been a central theme of reconfigurable computing research in both academia and the commercial world; and it continues to see significant attention since its such an important problem. Much of this work falls under the umbrella of *high-level synthesis*—in which an un-timed algorithmic description of behaviour is parsed by automated design tools which can generate corresponding, timed, hardware codes.

This has encompassed many different directions: including brand new languages and programming models tailored for general-purpose hardware development, domain-specific languages to generate hardware relevant to a specific application domain, and extensions and

compilers for existing, traditional, languages (primarily C/C++) for the generation of hardware codes. The latter is particularly enticing as it builds upon the existing knowledge-base of software developers and application code and is the focal point on the work in this thesis.

1.2 MOTIVATION AND CONTRIBUTIONS

FPGA technology offers a unique opportunity for accelerating computing. Inherently undefined, FPGAs have huge potential to realise performance and power consumption advantages over GPPs and alternative accelerators. However, unless the development challenges are addressed they will be relegated to a perennial niche.

Being able to even semi-automatically generate high performance accelerator hardware designs from existing high-level application software code, is certainly a respectable achievement that would open many doors for FPGA adoption. This vision captures the thesis' major motivation in this research area.

This thesis has two major objectives, that build upon this goal to aid accelerator design:

Accelerated data structures and HW/SW integration

This thesis investigates whether it is feasible to accelerate commonly used data structures in high-level application codes, using FPGAs within a reconfigurable computing environment. Using a graph algorithm as a case-study, the use of hardware accelerated priority queues is investigated.

- An important part to this work is the development and proposal of a concept and its implementation for the simple and transparent integration with application software.
- The spectrum of hardware-software co-design is investigated in this domain, with potential to spread the logical accelerator implementation across both devices.
- This type of hybrid, integrated, approach for hardware/software data structures has, to the best of my knowledge, not been explored in prior research. Therefore this topic constitutes novel work into tightly-coupled reconfigurable computing for real-world applications.

Polyhedral-assisted accelerator generation

This thesis investigates and proposes a novel methodology for the automated generation of hardware accelerators that target nested loop kernels, commonly found in numeric and scientific code. The generic approach is flexible and does not require loop nests to be tight. User input to the process is primarily the specification of a scattering function which dictates the behaviour and performance characteristics. Moreover, there is ample opportunities for design-space exploration and, possibly, automatic optimisation.

- Exploration into the current state of the art of high-level synthesis targeted to accelerate nested loop kernels, with a specific focus on existing imperative application code. Identify deficiencies and areas for improvement.

- Investigation and proposal of an innovative **architectural framework and hardware model** for the mapping of accelerator codes. Support for customisable **memory layout, pipelining, and tiling**. The significant novelty of this approach lies in the fact that FPGA hardware features are directly used without the basis on intermediate constructs. Even the closest previous approach differs significantly because it assumes a more simplistic architectural model. On the other hand, this work proposes a methodology to generate a tailored memory layout that improves distributed memory bandwidth across the design.

- Investigation of the use of the **polyhedral** compiler optimisation model as the basis for mapping kernels to the hardware model. Including the feasibility of using scattering functions for behavioural specification.

- Design and development of **techniques and algorithms for semi-automatically generating:** processing cores, performance-oriented memory layouts, interconnection and datapath network, and control schedules. Development and implementation of experimental toolchain PolyAcc.

- **Extensive experimental evaluation** of accelerator design permutations. Including different memory layouts, pipelined and non-pipelined processing cores, tiled modules, and multiple module designs.

These objectives tackle distinct ideas, but cut across the gamut of typical applications which are candidates for acceleration. Nested loop kernels are extremely important since they are often numerical algorithms with performance-critical requirements. Targeting general-purpose data structures is one way to tackle algorithms which are not necessarily numerical in nature, for example, graph processing, and which have had much less attention.

1.3 PUBLICATIONS

The following publications incorporate material from this thesis:

- Chandra, R., and Sinnen, O. Improving application performance with hardware data structures. In Proc. of the Intl. Symp. on Parallel & Distributed Processing, Workshops and PhD Forum (IPDPSW'10) (2010), IEEE, pp. 1–4.
- Chandra, R., and Sinnen, O. Towards automated optimisation of tool-generated HW/SW SoPC designs. In Proc. of the Intl. Symp. on Field Programmable Gate Arrays (FPGA'11) (2011), ACM, pp. 285–285. Abstract only.

1.4 THESIS STRUCTURE

This thesis begins, in Chapter 2 by examining reconfigurable computing in more absolute terms, include a brief review of contemporary execution platforms. Part of this includes a review of the various approaches to addressing hardware design challenges and a closer look at current state of the art research and commercial works focused on high-level synthesis. Here I introduce and discuss some of the distinguishing features of my suggested approach that will be developed in the remainder of the thesis.

Chapter 3 look at the first objective: accelerating a hardware data structure, specifically a priority queue. Firstly, I examine the data structure itself and then present a hardware implementation which has notably better theoretical performance than a software solution. Making the accelerator easily accessible from the software and host is vital, and a, transparent, Java-based implementation is discussed. There is a major caveat with the pure-hardware accelerator—it has

limited size scalability—but this is overcome with clever hardware-software partitioning.

The remainder of the thesis focuses on accelerators for nested loop kernels. This starts in Chapter 4, with a full exposition of the proposed architectural model. One of the foundations of the model is the underlying architecture of the FPGA device itself, which forms the basis of the design decisions here.

In Chapter 5 I introduce the polyhedral model and, with examples, show how it can map algorithmic input kernels to a space-time execution schedule that can be mated to the hardware model. Within this framework, techniques are presented to address exposing different degrees of parallelism, pipelining, and tiling.

Following that I present the collection of techniques and considerations for the actual accelerator generation, in Chapter 6. This includes algorithms for processing element generation, memory layout, interconnect, and control codes.

This objective culminates in an extensive evaluation, in Chapter 7. Synthesis and architectural characteristics, absolute performance, and qualitative design processes are presented and considered.

Finally, Chapter 8 concludes this thesis. We recap the outcomes of this work and reconsider them within the context of the thesis objectives. Furthermore, a number of directions and suggestions for continuing research is presented.

2 BACKGROUND

Facilitating easier usage of reconfigurable hardware is pivotal to the success of the technology in high performance computing. Research work is ongoing across different directions, primarily catering to different design abstractions.

This chapter presents a review of reconfigurable computing systems, providing context of the execution platform considered in the remainder of the thesis. This includes the architectures, components, and design methodology. The latter is the focus for a thorough examination of existing and ongoing research, categorised into three areas: techniques to simplify and enhance hardware design, automatic hardware generation from application software, and the advances of software and compiler design (which cuts across the field). Each category targets rather different goals but all are equally important research areas. Finally, this culminates by reflecting on the past work and presenting an overview of this thesis, and how it gels with this body of work.

2.1 RECONFIGURABLE COMPUTING SYSTEMS

Let us begin by introducing a few reconfigurable systems to examine trends in architecture.

One of the simplest ways to build a reconfigurable computing platform is by attaching a FPGA development board (or 'acceleration module') to a standard GPP workstation using a common I/O interface. This could be as simplistic as connecting a standard FPGA development board via USB. More typically, PCI Express (PCIe) development boards have gained a lot of popularity. Such boards have been in

production since the early 1990s [44]. One, or more, boards can easily be installed into a workstation and server chassis and the PCIe interface provides relatively high system bandwidth, with strong platform support given the universality of PCIe.

The PCB design can play an important role for the focus of the system and, moreover, highlights the flexibility of the add-on board approach. For example, typically many memory modules can be included thus providing substantial concurrent memory bandwidth [44], or it could be focused on external I/O interfaces and signal capture, or even include multiple FPGAs.

Taking a quite different approach, XtremeData developed the XD1000 [108] system around the AMD Opteron platform and built using a dual-socket server motherboard. One socket is populated with a traditional AMD Opteron 248 processor. But the other socket is populated with an accelerator module that features an Altera Stratix II EPS2180 FPGA device, 4MB of SRAM memory, 32MB flash memory, and support for utilising the dedicated socket's DDR memory channel. The two sockets can communicate via a HyperTransport communications channel that provides a peak theoretical bandwidth of 3.2GB/s with a, relatively, low latency. Both sockets have access to dedicated DDR memory banks.

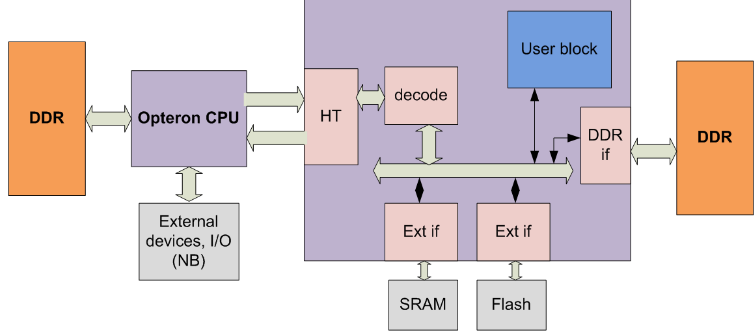


Figure 1: XD1000 development system base configuration architecture overview.

Figure 1 shows a simplified block diagram of the system architecture including conceptual details of the XtremeData system and FPGA configuration [107]. The base system can be extended by incorporating custom 'user defined' components that sit on the main internal bus and can thus participate in communication with the external world. The HT decoder correctly selects the appropriate com-

ponents based on the specified addresses of the HT communication. In this system there is no cache-coherence or ability to share memory, unfortunately, so all communication is explicit.

For large systems, scalability and modularity is a significant concern. For instance in the BEE2 project a mesh node consisting purely of FPGAs, five in total—four for processing and one for control, was used and networked using a complex multi-link mesh connection scheme [29].

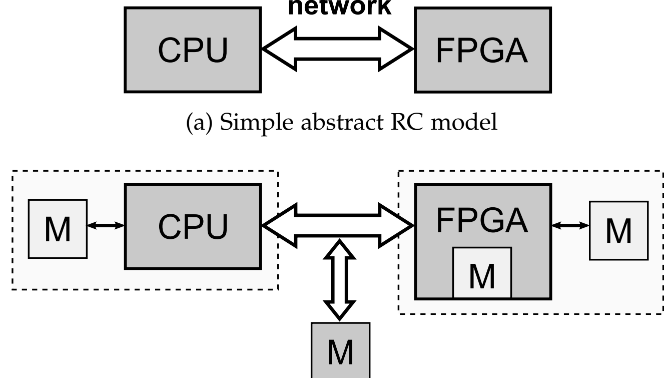
To achieve even greater performance for 'reconfigurable supercomputers,' vendors have connected multiple CPU/FPGA processing nodes together to form tightly-coupled processing clusters. For example the SGI RASC systems follow a traditional blade server architecture but with FPGAs in half the sockets [89]. Moreover, this is complemented with a proprietary network system to provide much better performance than, say, commodity Ethernet. Software support for accessing the global memory from the FPGA is claimed.

The SRC computers like the SRC-6, are similar in philosophy. This system is based on their proprietary MAP module system, where a MAP FPGA processing module–which consists of two user FPGA, one control FPGA, and memory units (a Series E module)—is connected to the MAP system interconnection backbone along with other MAP FPGA or CPU modules [60]. This setup isolated the FPGA modules from the GPP units locally, instead modelling them more purely as dedicated processing resources.

2.1.1 *RC system model*

The foundation of this work is sculpted by the topology and architecture of contemporary reconfigurable computing (RC) systems. As with most computer technologies, these are evolving at a rapid pace. Given the complexity and propensity of change, we abstract the system by capturing it in a simplified general model which simplifies analysis.

At the core of the model, a system consists of a GPP and its support structure, associated memories and interfaces, which is mated with an FPGA device across a communications channel or network as shown, in its most basic form, in Figure 2a. The two devices can communicate each-to-the-other which facilitates the transfer of data and workload sharing. Memory is, of course, vital to any computing



(a) Simple abstract RC model

(b) Model extended to show typical memory placements

Figure 2: Basic reconfigurable system model.

platform and there are numerous ways this can be designed. Figure 2b captures a few of the common memory layout topologies within this model. Ultimately this resolves to a familiar shared-memory or distributed-memory approach, or possibly a combination of the above. One of the unique characteristics of FPGAs is the customisable embedded memory available. Although small in quantity, it is devoid of external latencies and certainly the fastest available memory to be used for FPGA computation, irrespective of the surrounding memory architecture.

In more exotic custom implementations it is feasible that even tighter-coupled architectures are possible, for example, the FPGA as internal 'CPU peripheral' or a dedicated co-processor [100]. But this is not feasible when using commodity hardware (x86 processors) and platforms, hence this technique is typically reserved to embedded system applications.

This system model can be extended further by considering this CPU/FPGA mesh as only a single node within an even larger processing network. Nodes can be connected together in a network, maybe in a traditional bus or ring topology, for increased scalability. However, as our platform spreads further–whether as multiple machines on a private network to a wide-area, or Internet, distributed system–we must be aware that bandwidth becomes reduced as the cost of communication increases due to the extra overhead and complexity involved. In practise there have been a number of minor differences between system architectures in both research and literature but my generalised model encompasses them well as a working approximation.

2.1.2 *Computational characteristics*

Before looking at practical reconfigurable systems let us characterise the main processing components–the CPU and FPGA. Flynn stipulated that the design of a computer can be classified based on the flow of instructions and data [38]. He developed a taxonomy based on these factors shown in Table 1.

	Single Data	Multiple Data
Single Instruction	SISD	SIMD
Multiple Instruction	MISD	MIMD

Table 1: Flynn's taxonomy.

Based on this model, a classic GPP can be classified as a SISD device. However, most processors typically also include a SIMD 'component', for example MMX/SSE units in Intel's Pentium and Core architectures, AltiVec unit in PowerPC architecture, and the utilisation of a pipelined architecture. FPGAs on the other hand are completely undefined and thus can implement any taxonomy making classification unreliable. But as the advantage of FPGAs are in their ability to implement parallel constructs, typically a SIMD or MIMD architecture would be employed depending on the application.

With the advent of multi-core GPPs they can also be classified as MIMD, depending on how they are used.

Memory type	System (e.g. DDR2)	Board (e.g. DDR2)	Embedded M144K	Embedded M9K
Bandwidth (aggr.)	16	8	200	3000
Bandwidth (each)	8	4	4.3	2.4
Channels	2	2	64	1280
Capacity (each)	2-8 GB	2 GB	0.1 M	0.009 M
Capacity (total)	4-32 GB	4 GB	1.1 M	1.4 M

Table 2: Approximate comparison of memory types within a reconfigurable system. Bandwidth is in GB/s.

MEMORY ARCHITECTURE Furthermore, Flynn's taxonomy fails to consider memory architecture which is an integral part of a computing system. In the case of the GPP it is a shared-memory architecture—each core communicates through a common memory pool—although technically the hierarchy is much richer with layered caches. The FPGA memory architecture is again undefined. Individual processing elements employ their own small, independent, distributed memories, like scratchpad memory, but globally it can also be classified as a shared-memory architecture. But there are many levels of parallelism that can be employed, for instance as processing nodes are scaled to large distributed systems, a distributed memory architecture could be preferable which utilises a message passing model of communication.

Table 2 attempts to categorise the differences in available memories. In this table, 'system' refers to a typical x86 workstation, while the remaining classifications reflect—as representative example—the Terasic DE4 development board [98] with Altera Stratrix IV FPGA (EP4SGX530). In this comparison its quite easy to see that FPGAs have the potential to achieve two orders of magnitude better bandwidth performance by utilising the large number of, relatively fast, small embedded memories. On-chip memory capacity, however, is very small by comparison; lending to the importance of well designed memory management (including tiered memory hierarchies).

DE4 is a PCIe-based FPGA add-on board.

INTERCONNECT TECHNOLOGIES Designers are still faced with the challenge of getting data into (and out of) these memories in the first place, and particularly during computation. Ultimately in most configurations a single communication link is the basis for data transfer between the software domain (CPU) and the hardware domain (FPGA). For this reason communication and interconnection is of vital importance to the performance of reconfigurable systems over a range of workloads.

A contemporary x86 CPU operates between 2-3 GHz where as an optimised hardware design on an FPGA may only operate at 200-300 MHz—an order of magnitude slower! FPGA-based designs must perform significantly more work per clock cycle to compensate for the comparative frequency disadvantage. Moreover, there is a compounding cost associated with each communication, or data transfer, between FPGA and the CPU which typically increases the further the FPGA is from the CPU. The overall communication impact across

various RC configurations are also adversely affected by auxiliary factors including bus loading, system load, OS drivers, and software stack, which makes an accurate practical comparison, as opposed to theoretical maximums, a difficult feat; but Table 3 presents some approximations.

Connection type	socket	add-on card (PCIe)	proprietary (e.g., SGI Numalink)	Ethernet (10Gb)	I/O (e.g. USB3)
Bandwidth	4-50	4-16	2-4	1.25	0.625
Latency	very low	low	moderate	high	moderate

Table 3: Comparison of interconnect technologies used in reconfigurable systems, bandwidth in GB/s [104].

2.2 DEVELOPMENT FOR RECONFIGURABLE SYSTEMS

It has been established that well designed reconfigurable systems can offer extremely attractive performance and power efficiencies, when compared to alternative technologies. But, the uptake of such systems is limited by development barriers [22, 51]. Developing for RC is non-trivial due to the numerous ad-hoc interfaces and specifications arising from lack of unification at the platform level. On top of this there is a lack of transparency and inter-operability between design languages, methods, and techniques used in the configuration (or programming) of the various components.

A broad knowledge of software development, parallel programming techniques, and hardware design is required for successful hardware acceleration. Figure 3 shows the typical design process from application specification to source code ready for final software and hardware compilation. Unlike in traditional sequential GPP development, parallel and distributed programming paradigms must be used.

This entails analysing the original application in terms of sub-tasks, or chunks of atomically executable code, which can then be ordered by their dependences. The next stage is to identify and characterise the sub-tasks (in terms of the nature of their execution) in order to attribute them to appropriate computing resources, including FPGAs.

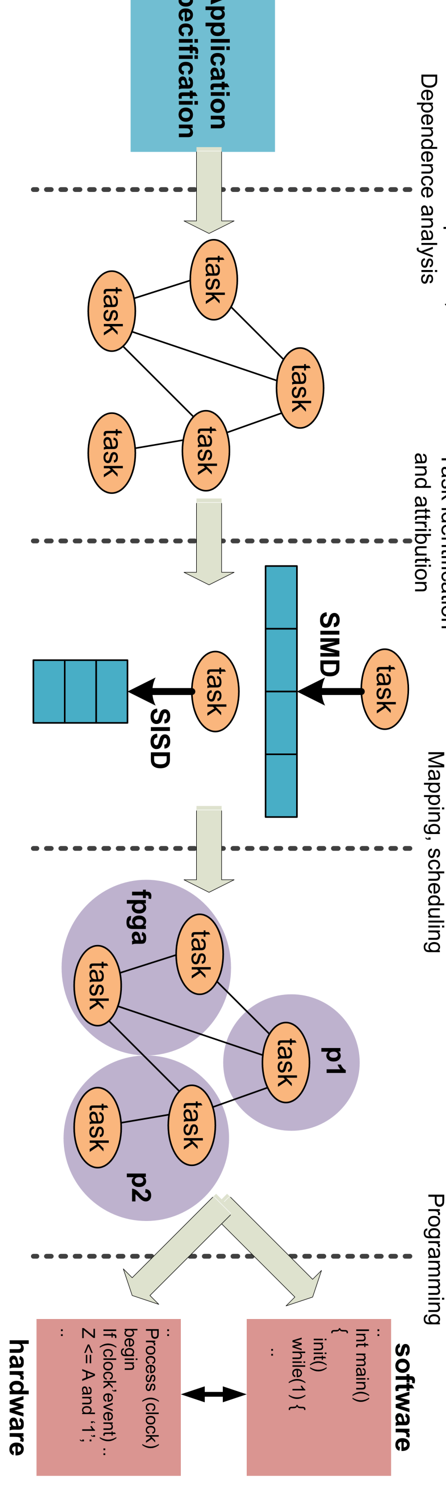


Figure 3: A typical design-flow for reconfigurable system development

Since FPGA architectures are inherently undefined, it is a non-trivial problem, introducing a very challenging facet to traditional parallel programming scheduling approaches. This can be a re-iterative process as the cost/benefit of hardware allocations are often difficult to predict without feedback from experimental results. The final step is code generation. Adapted source code for the GPP to capture the task schedule and implement necessary communication and synchronisation mechanisms necessary for interoperability. Hardware description for the FPGA to define the appropriate target architecture and implement the corresponding communication and synchronisation mechanisms.

In practice, designs cut across this overall flow with different approaches to each phase. Moreover, there are a range of different levels of abstraction that can be employed. The next sections explore some of the research in this area. Firstly I will present the latest trends in hardware design; centred around languages and tools for architectural specification. Then I shall take a broad look at the software perspective—parallel programming techniques common in traditional high-performance computing. Finally looking at work that attempts to meld these areas together for the purpose of reconfigurable computing, and their successes and areas of further opportunity.

2.3 HARDWARE DESIGN LANGUAGES

Hardware description languages (HDLs), such as VHDL (Very-high-speed integrated circuits Hardware Description Language) and Verilog, are currently the most common means of developing hardware designs. These languages are generally considered to be used for the *capture* of hardware architectures at varying abstractions. The abstractions can range from low-level gate specification, structural and register-transfer-level (RTL) specification, and to even higher-level behavioural descriptions. These hardware descriptions are then synthesised, a form of compilation, to generate a netlist that can be used for device programming.

A netlist specifies the exact configuration of each hardware element that makes up an FPGA. Thus it is a structural representation of the overall hardware design.

HDLs are programming languages but there are important differences between them and the traditional application languages used for software development. While most traditional languages are procedural with limited language-level support for concurrency and with no notion of time, HDLs can easily express multiple parallel pro-

cesses, concurrent synchronisation techniques, and have strong notations for accurate timing. Thus moving from traditional software development to designing hardware for HDLs represents a paradigm shift in approach and considerations.

For this reason a number of works in academic circles and commercial offerings have attempted to bridge this gap. One popular approach appears to be bringing the syntactic familiarity of traditional high-level programming languages to the domain of hardware description. This is motivated by the premise that it can reduce development cost and time by leveraging the language competence of the software developers, combined with the efficiency gains of working at higher abstractions. C and C++ are popular choices due to their traditional popularity in low-level software designs.

Mentor Graphics offers the Catapult suite for accelerated hardware design, based on the ANSI C++ language [70]. C++, combined with some specific hardware-centric annotations, is as a language to describe the *hardware functionality* (as opposed to the software or algorithmic functionality). The tool can then generate corresponding RTL code, ready for synthesis. JHDL [16] has been developed at Brigham Young University and is a Java-based suite to meet the needs of hardware designers–streamlining the hardware capture, verification, and synthesis stages of hardware design—using a Java-based object-oriented design to describe the hardware. Another open-source package is FpgaC which is again based on a subset of C and aims to provide a higher-level means of describing general-purpose hardware architectures [40, 94]. Handel-C from Celoxia is again developed from a subset of C with extensions to aid the instantiations and expression of parallel hardware structures [102] though it is intended to be able to capture more algorithmic details.

But such efforts have also come under criticism in literature: can languages based on C, originally developed in the 1960s for procedural, imperative programming be the right choice for hardware design? It can be argued that it is not necessarily the most appropriate and intuitive choice; C, and similar high-level languages, have been developed with the goal of developing applications for a specific class of von Neumann computers, which are quite different to the underlying reconfigurable fabric present in an FPGA device. Hardware architectures are more closely aligned with functional characteristics and deeply rooted in concurrency.

[93] states that the problem is due to the mismatch of expressiveness between the synchronous, concurrent processing capability of the hardware and the language which is termed as the *semantic gap*. They propose library-based extensions to an existing OO language as the basis for a new programming model that captures the asynchronous and parallel nature of the hardware better. While such an effort is an improvement by building a higher abstraction to hardware design it is still a marked departure from using the base language in the context of traditional algorithmic and application development.

Single Assignment C (SA-C) was developed at Colorado State University as part of the Cameron project [49] and is popular in the literature. Again, the lack of specific ways to address wanted hardware features and in particular the ability to represent coarse and fine grained parallelism were motivating factors in its development. It was originally targeted for image processing applications and this is clear with support for multi-dimensional arrays and windows for example. Parts of the C language have been removed including recursion and pointer operations and other parts added. Ultimately such an approach seems far removed from both software developer familiarity and hardware designer familiarity, which raises the question of why even start from a C, von Neumann directed, derivative language at all?

This notion is evident in literature and, for example, a shift away from traditional HDLs and procedural programming languages towards a *function-programming* paradigm is mandated in [17]. Their approach is two-pronged; firstly the development of a new intermediate language, CASM, and secondly the development of functional programming language compilers to target the generation of CASM modules. CASM generates hardware models based on a C-like algorithmic state machine description including support for levels of synchronisation and recursion and is based on a token-paradigm where operations are triggered by data presence.

Functional programming is again presented as a better way to represent implicitly parallel programs in [91]. As an alternative approach, they also propose the incorporation of join patterns that provide a means to describe synchronisation across parallel activities at a raised level of abstraction. The proposed implementation strategy was a library extension for traditional high-level languages, like C# and Java. The PARO system [50] is a recent example of the popularity of func-

tional programming concepts. This work is based on their own new custom functional semantics tailored to dataflow dominant applications. More recently, [11] presents a Scala-based domain-specific language that attempts to meld some of the functional techniques with object-orientation, and advanced typing that is typical of Scala.

2.3.1 *Architecture description languages*

In more recent times there has been some exploration of architecture description languages (ADLs) for accelerated hardware design. This language class has been developed with the intent to abstract away lower-level parts of typical HDL descriptions. This effort originated from the need to aid processor design, particularly for simulation and instruction set exploration.

Current ADLs can be further separated into an instruction-set centric branch and an architecture centric branch, and of course there is some work that attempts to combine both. Instruction-set centric languages like nML [52] provide a programmer's view of the architecture by exposing the instruction set which is particularly useful in a rapid simulation and design environment–while ignoring cycle-accuracy and architectural details. Architecture-centric languages like MIMOLA [71] focus on the structure and connectivity of components which is advantageous for synthesis and tool generation and enables a much richer architectural description for more accurate simulation, albeit at a greater complexity and simulation performance cost.

Hybrid instruction-set and architecture oriented languages [52] like LISA and EXPRESSION bridge the gap between the two domains by taking a 'best of both worlds' approach. LISA has since been commercialised and developed by CoWare as the Processor Designer tool. This product has the capability to take LISA hardware descriptions of both accelerators and fully-fledged CPUs and synthesise it for hardware implementation, by RTL generation, and generate a complete supporting software tool chain specific to the instruction-set ready for application development.

2.3.2 *System-level design*

Taking an even higher-level view of the demands of hardware design we are led to *system-level* design methodologies and tools. SpecC and

SystemC are two such languages both based on C/C++ with extension libraries and macros. These tools are referred to as system description languages (SDLs) as they aim to describe not just the hardware but the entire application compute ecosystem—including the software[1], communications channels, and the hardware. This represents a high-level modelling technique and is useful for rapid prototyping and design verification.

However, consequently upon completion of such an approach, designers are faced with the task of implementing the modelled system and thus are left again to implement hardware using HDLs and, by extension, re-verification for the underlying technology. For this reason the attractiveness of system-level design is diminished to some extent but there is work being undertaken to address these concerns by automatically generating hardware descriptions from SDL components. Forte's Cynthesizer is a commercial package that generates optimised RTL HDL code from the system-level description [39]. Cadence's C-to-Silicon compiler claims to perform much the same tasks and, again, slashing development times and errors with high-quality generated RTL [23].

Other academic research has developed more formal methods of system modelling. One such work is the development of SystemJ [47] which is a language that combines the expressiveness of traditional Java with the synchronisation and reactivity of ESTEREL [21] in a hybrid Java environment. SystemJ is based on the globally asynchronous, locally synchronous (GALS) model. The greater system environment can be composed of multiple clock-domains that can communicate with each other by send and receive functions using *rendezvous*. Each clock-domain consists of synchronous reactions which can communicate using signals. SystemJ has been designed for embedded systems although it can run at desktop-level via environmental emulation. At present work is being conducted to improve the transparency of compilation.

1 It must be noted there is an overlap with the software design domain, as system-level tools can be used to describe both hardware and software, but they are largely ignored by the pure software community.

2.4 APPLICATION SYNTHESIS

Successful reconfigurable computing development must meld both a robust hardware architecture and optimised parallel software application and algorithms each-to-the-other to efficiently harness the available resources. The previous section focused on a "bottom-up" approach to acceleration: improving and developing hardware design using domain-specific tools and languages. Application synthesis represents a "top-down" approach where designers can move from the application context and, with the help of appropriate tooling, generate a hardware accelerator.

This combination of hardware and software is non-trivial and it can be argued that it will be more difficult to extract the same level of low-level hardware performance given the extra abstraction layers involved. On the other hand, the application-level may lend better visibility to the desired behaviours and outcomes and make it more accessible for developers to understand (and direct) how its execution can be accelerated by the hardware. This design effort raises three important issues: 1) identifying the source code blocks which are the best candidates for hardware implementation, 2) interface and synchronisation between host (software) and accelerator (hardware), and 3) maintaining application integrity while recognising and utilising hardware resources.

I now review some of the research in this field, classified into technologies that either target low-level applications or high-level applications. Firstly, the distinction between low-level and high-level applications must be explained. Low-level applications are typically developed in C-like languages that provide low-level access to hardware components present in the system design. In essence, the application is written close-to-the-metal, prioritising performance at the expense of abstractions and generalisation. Low-level applications often consist of computationally-heavy loop nests, for example in scientific or numerical algorithms, which have traditionally been the primary target for acceleration. On the other hand, high-level applications are more reminiscent of modern-day software application; they are typically developed in languages providing greater abstraction and place emphasis on modularity and extension. Performance is, of course, very important. However, so is flexibility, reliability, and portability.

2.4.1 *Low-level application synthesis*

A number of works in this area are based around traditional high-level languages with extensions or libraries to facilitate the incorporation of hardware modules within the application design. For example, the ImpulseC toolchain from Impulse Accelerated Technologies extends C/C++ to support *stream-based* parallel programming constructs that can be synthesised in hardware [53]. For best synthesis results, applications should be developed, or modified, based on the communicating sequential processes (CSP) programming model. Currently the tool can target both embedded systems, which might feature both custom hardware and CPU on the same chip (SPoC solution), through to high-performance reconfigurable computing systems with multi-socket FPGA and CPU like the XD1000. Mitrion-C from Mitrionics is another C/C++ based language, however, it targets the generation of customised soft-processor cores rather than RTL modules. The language itself is even more of a distinct departure from traditional C as it is implicitly parallel, traditional statements are treated as expressions, it has new keywords, and has a modified type system and data containers [73].

Coming back to traditional software applications there has been work and commercial tools that do not require substantial application modification. The Altera Nios C2H compiler is one such package which generates hardware accelerator peripherals for their custom Nios II soft-processor environment. This tool supports ANSI C features including pointers and features novel techniques to handle memory accesses and latencies by generating multiple unique memory ports. However, they have noted the effectiveness of the tool can be substantially improved if the source code is optimised through feature restriction and manual consolidation to make it more easily analysable [4]. I present a more thorough examination of this tool in Section 2.5, albeit this tool will be discontinued in 2013 in favour of an OpenCL approach, see below.

[48] presents the ROCCC synthesis framework for accelerator generation. It follows a familiar compilation flow with a front-end implementing compiler transformations that generates an intermediate description for later hardware generation. A special "Smart Buffer" is proposed to handle data communications that can coalesce and man-

age dataflow in certain cases, for example, windowed operations, or with explicit annotations.

The flexibility and accessibility of the LLVM compiler [63] has led to the recent development of new tools for hardware generation. These works include C-to-Verilog [85], xPilot (later commercialised as AutoESL [109], which is now part of Xilinx Vivado [106]), and the LegUp [24] compiler. The LLVM front-end develops a low-level, but typed, intermediate representation (IR) from the input code that is very amenable to optimisation and transformation. A back-end code generation process can, later, transform from the IR into targeted platform-specific codes. These works leverage this framework to: 1) create new back-ends that generation HDL code, and 2) supplement this process with a set of hardware-specific optimisation passes.

Recently, Altera have been working on an SDK kit for the acceleration and mapping of Open Computing Language (OpenCL) codes to FPGAs [7]. A similar direction has been explored in [54]. OpenCL is a framework for heterogeneous application execution, although it has historically been pitched as an alternative to CUDA for GPU programming. OpenCL has begun to penetrate mainstream development circles, because of the desire to leverage GPU acceleration by applications. Programs must be modified to fit within the specific OpenCL kernel paradigm, and therefore already capture vital information for the acceleration of code which can be exploited in OpenCL-to-FPGA approaches.

2.4.2 *High-level application synthesis*

Less attention has been paid to the synthesis of higher-level applications for complete or partial FPGA implementation, however, some interesting approaches have been presented.

A common strategy typically for applications involving numerical methods has been the compilation of graphically described algorithms. These are based around using graphical building blocks that are cascaded and connected together like in a system diagram. Starbridge's Viva product is based around this paradigm which they have adopted as they believe C-based languages are not ideal for FPGA development. Matlab's SimuLink environment can be used in much the same way in-conjunction with tools like Altera's DSP Builder that can compile blocks into optimised RTL code [6].

Going back to a traditional software application, Kansas University has proposed the HybridThreads toolchain which allows programmers to develop applications using traditional threaded models [95]. The threads are then used as the abstracted building blocks for implementation on a reconfigurable computing system. The toolchain compiles threads which can then be run on either the GPP or dedicated hardware resources on the FPGA. Operating system constructs play an important role in the communication, synchronisation, and scheduling of the system operation. Currently the implementation work is still in progress.

Another technique gaining popularity for hardware compilation is byte-code analysis which can work independently of the application source. By orienting the hardware compilation process on the already compiled byte-code representation of an application the development team need not change their approach or rewrite their algorithms. The drawback of the approach is that the visibility of the initial application structure, and more importantly knowledge of the *algorithmic intent*, is reduced making it potentially more difficult to reach the performance ceiling of the hardware. Works that have taken this approach include [72, 96], other very similar but restrictive work includes [25], which places some limitations on the possible byte-codes used, and hence the input source, and [101] which relies on source to be written according to a threaded CSP programming model.

2.5 CASE STUDY: ACCELERATION USING C2H COMPILER

I believe application synthesis has a pivotal role to play in hardware acceleration. The merits of this abstraction can be debated from a theoretical and modelling perspective, but the popularity of C-like languages for algorithmic description is undeniable, making it an extremely attractive target. Altera's C2H Compiler [9] is a commercial production-grade tool that can be used for this purpose, and this section explores it in greater depth. Figure 4 shows the overall design flow for accelerator generation.

The C2H Compiler works in-conjunction with the Altera Nios II soft-processor tool-chain. Applications are written in C/C++ and execute on the Nios II CPU with access to the complete SoPC environment by way of the Avalon bus. Numeric kernels and/or processing loops can be flagged for hardware implementation from within the

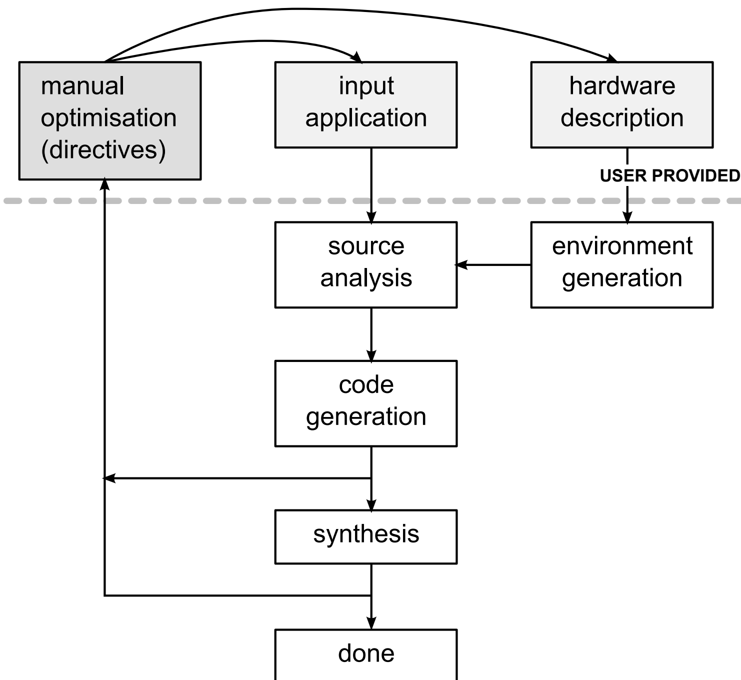


Figure 4: Application synthesis design flow using the Altera C2H tool.

accompanying IDE, marketed as "right click to accelerate" technology. The C2H compiler then examines this kernel to generate an external hardware accelerator along with the necessary plumbing to incorporate it within the SoPC. Finally, the application build process automatically utilises the new accelerator. Automated hardware generation is hard and the C2H tool can not perform magic. Instead it follows a few generic principles, from [9]:

- *Direct mapping between C constructs and hardware structure*. The mapping rules are defined and fixed making for predictable and consistent hardware generation. For example, mathematical operators become equivalent hardware circuits and loops are generated as state machines. This also demonstrates the compiler's inability to extract meaningful information about the application's intention.

- *Memory accesses are generated as ports to external memory.* All accesses to memory via arrays, or pointer de-references, are analysed and converted to dedicated memory ports. This has the advantage of supporting parallel pipelined memory communications. However, this is limited by the number of physical memories and the data layout within them.

- *As-soon-as-possible scheduling applied to independent statements*. Statements which have no dependencies are executed early in an attempt to maximise parallel execution.
- *Sub-functions are generated as a shared hardware resource within the accelerator*. They are automatically parsed by the compiler and generated appropriately. These components are shared and thus re-used over multiple loop iterations, for instance.

2.5.1 *Complete SoPC example*

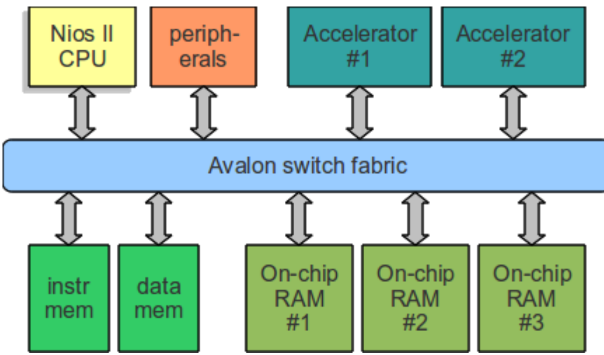


Figure 5: Example SoPC environment built around the Altera Nios II CPU and Avalon interconnect.

Figure 5 depicts a completed RC system. It incorporates a Nios II GPP along with memories and standard peripherals. Two accelerator modules are featured and each utilises dedicated on-chip memory devices to provide increased levels of bandwidth, critical in improving performance. The system bus is automatically generated based on interface specifications. The software part of the equation is not shown but the standard software method invocations are automatically adapted with functionality to engage the hardware accelerators during run-time execution.

2.5.2 *Optimisations and compiler directives*

Different code formulations can result in significantly different hardware output, and thus the intention of the source modifications is to automate some of the optimisations made by a designer. While traditional compiler optimisations are relevant they may not always be directly applicable. Increasing local memory bandwidth (data avail-

ability) typically provides a noticeable improvement in performance, because acceleration kernels are targeted to work best with streaming data. On an FPGA this can be achieved by adjusting the number and type of local memories and to keep the contained data as relevant as possible.

Consider the following example function to demonstrate the benefit of a simple fine-grained optimisation:

```
void foo(int *a, int *b) {
  int i, t;
  for (i=0; i<N; i++) {
    t = *(a+i);
    *(a+i) = 12 * *(b+i);
    *(b+i) = 7 * t;
  }
}
```

This C function is an arbitrary example that performs computation on two independent arrays and then swaps the values. When analysed by the Altera C2H compiler, an accelerator featuring two multipliers and two memory ports is generated and controlled by a state machine that follows the input algorithm. However, the independence of the two arrays is not detected, due to the inference of *t* as a read and write dependence, and thus it could not be pipelined. This is easily alleviated by applying a scalar expansion of *t* along with loop splitting [59]:

```
void bar(int *a, int *b) {
  int i, t[N];
  for (i=0; i<N; i++) t[i] = *(a+i);
  for (i=0; i<N; i++) *(a+i) = 12 * *(b+i);
  for (i=0; i<N; i++) *(b+i) = 7 * t[i];
}
```

This change, while simple to implement, removes the dependence which convinces the C2H compiler to generate fully pipelined hardware. It is beneficial to be more explicit about the nature of the memory configuration, if possible. For example, if we have access to three independent memories within the system it can be explicitly attributed to variables by prefacing the function with `#pragma` directives, as follows:

```
#pragma altera_accelerate unshare_pointer bar/a
#pragma altera_accelerate unshare_pointer bar/b
#pragma altera_accelerate unshare_pointer bar/t
#pragma altera_accelerate connect_variable bar/a to mem_2/s1
#pragma altera_accelerate connect_variable bar/b to mem_1/s1
#pragma altera_accelerate connect_variable bar/t to mem_0/s1
```

Managing memory hierarchies, at the different levels of a RC system, is one of the open challenges of HLS tools [35]. It has been shown that leveraging on-chip memories can potentially significantly improve accelerator performance by virtue of greater memory bandwidth through faster access times and, moreover, the possibility of multiple data accesses when utilising multiple memory ports. However, it is already clear that it is not easy to wield this power in a meaningful way—manually specifying the memory layout, as above, is not a satisfactorily scalable solution. Moreover, one of the major drawbacks with the C2H Compiler is that these memories must be explicitly created. Each additional memory must be specified ahead-of-time in the accompanying SoPC Builder tool, a significant hurdle for rapid design-space exploration.

Looking at another example, matrix multiplication is a common and important function used within many algorithms. Again, manual optimisations can be applied to a typical procedural implementation that can be used to improve performance while balancing resource consumption. In particular, we can use a scalar variable to ensure the write-operation in the inner-loop is kept in local storage (a register), and can then employ local memories (added to the system and then specified during compilation with `#pragma` directives) to provide greater memory bandwidth. Thus, benefiting from unrolling the inner-loop and performing parallel multiplications. For example:

```
void mat_mult(int *A, int *B, int *C) {
  int i, j, k, t;
  for (i=0; i<DIM; i++) {
    for (j=0; j<DIM; j++) {
      t = 0;
      for (k=0; k<DIM; k+=2) {
        t += ARR(B,k,j) * ARR(A,i,k);
        t += ARR(B,k+1,j) * ARR(A,i,k+1);
      }
```

```
      ARR(C,i,j) = t;
    }
  }
}
```

In this case ARR() is a macro for a pointer-based memory access, preferred by the C2H compiler over array indexing.

2.5.3 *Impact of manual optimisation*

Combining the aforementioned optimisations I present some results of the impact on the final performance. Three case studies have been explored: 1) the arbitrary array swap computation, with array length of 8192, 2) integer matrix multiplication implementation using 36864 element square matrices, and 3) a 3x3 stencil filter from an image processing application. Each example has been implemented in C and ran as pure software on an Altera Nios II (fast-grade) soft-CPU, then once again after employing the C2H compiler to generate an out-of-box hardware accelerator for each function, and finally after applying code-optimisations to the input function and then re-generating the hardware components with the C2H compiler. All experimental results were obtained while running on an Terasic Altera DE2-70 FPGA development kit featuring a Cyclone II EP2C70F896 FPGA device. All problem sizes in this experiment fit within the on-chip memory capacity of the FPGA; performance scaling with respect to problem size has been tested and was in-line with the presented results, but are omitted here.

In Figure 6 we can see that there are substantial benefits to performing source-code analysis and optimisation as proposed in this work, and a reflection on the potential of C2H tools. The out-of-box speedup is greater for kernel (3), the image filter, relative to the other examples due to the number of memory accesses in the inner loop which can be partially pipelined.

On the surface it seems odd to think of using automated application synthesis tools, like the C2H Compiler, as being a manual process. But, these results indicate that it is vital to employ manual optimisation to achieve significant performance improvements compared to software-only implementations. [76] presents an FFT case study,

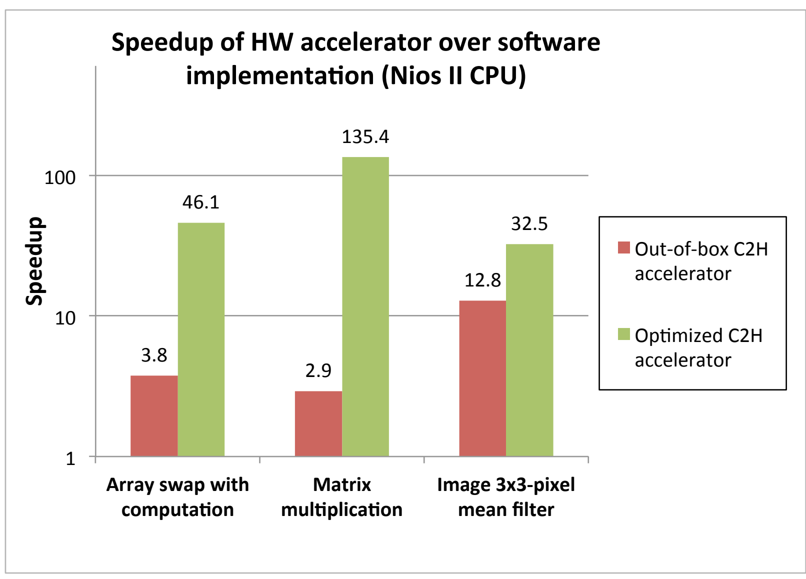


Figure 6: Performance impact of manual code optimisations and directives to input prior to use of the C2H Compiler generation process.

based on the competing, but more advanced, AutoPilot tool, which highlights many of the same considerations.

Furthermore, an easily overlooked side-effect of this methodology is the substantial time spent in the iterative cycle making and testing manual code modifications. Such changes are not strictly necessary except for non-trivial applications, but when working with performance or resource constraints, as often the case, the developer must undertake this challenge and it has been shown to make a significant difference at the cost of engineering effort [64].

automated tool-specific optimisations A possible approach to make some of these tools more viable could be to augment them with a framework for the automatic application of compiler directives and code modifications that would otherwise be performed manually. This approach was an early idea proposed by the thesis author in [28]. Techniques such as auto-tuning [99] can play a significant role in being able to achieve this, by automatically trying different permutations based on heuristics.

2.6 polyhedral compilation

The holy grail for (software) compilers is to automatically extract performance gains from the existing application code. Current compilers

have evolved to become aggressive and competent at this. To a certain extent, they can extract parallelism from both procedural and parallel application descriptions, but this is highly dependent on the specific algorithm or input codes.

The polyhedral model is a popular approach to address some of these fundamental compilation challenges for numeric and scientific codes. It is a mathematical framework for the analysis and optimisation of loop nests. It stems from seminal work conducted in [58, 62] and then applied to the generation of systolic processor arrays from computations expressed as recurrence equations [61, 82]. Each iteration of a [recurring] computation falls within an iteration polyhedron. A transformation can be applied to each iteration from this space to a different polyhedron with temporal and spatial dimensions, thereby defining an—ideally, parallel—execution schedule. Other compiler advances in the form of dependence analysis and loop extraction have enabled it to be applied to the parallelisation of general loop nests expressed in traditional procedural languages. Polly [46] is a modern framework implementation built upon the LLVM compiler. Scattering functions can be provided manually, or automatic selection can be attempted using the Pluto polyhedral optimiser [20].

Most work has focused on this from the context of large-scale multiprocessor systems as the target platform, which more or less follow traditional PRAM-based models. This is in stark contrast to the original works which used the techniques in the context of VLSI systolic arrays with a data-flow model of memory transfer [61, 82].

Utilising polyhedral analysis for the purpose of hardware, while not new, is a recent area in which a few works have explored with some success. In [50] polyhedral space-time mapping techniques are used in hardware synthesis with the emphasis on the partitioning with tiles applied to a more conventional processor array. Albeit, they abandon procedural input languages and propose the use of new functional languages to better express parallelism. A polyhedral basis is featured in [37] where an existing polyhedral code generator is extended to generate HDL control code for loops. They emphasised data locality, assuming a traditional singular memory. Parallelism is inferred using auxiliary loop processing techniques rather than being explicitly described in the schedule. Likewise, the polyhedral approach was used in [2] to generate an optimised schedule for feeding data to a single floating-point processor. The processor was

application-specific, generated externally, and fully-pipelined; therefore, the objective was to minimise datapath stalls.

Works such as [68] apply polyhedral techniques to generate on-chip memories for reused data items, while minimising the on-chip memory requirements. [15] develops a memory controller for external SDRAM that maximises bandwidth through intelligent addressing. In contrast, I focus on exploiting multiple on-chip memories to their fullest for greatest throughput. [33] presents an ILP-based automatic memory partitioning method optimised for both throughput and, uniquely, power usage. They partition data in blocks or according to a cyclic schedule. The memory architecture is based on multiple banks but behind a unified interface handling decode logic.

2.7 OVERVIEW OF THIS THESIS

From the preceding review of state and application of reconfigurable computing, there are two key direction which will be pursued: 1) the treatment of high-level application codes that can not benefit from typical optimisations as with nested numeric kernels and 2) the melding of state-of-the-art polyhedral techniques to generate computationally-heavy accelerators. Between them, these two distinct areas encompass a wide-gamut of applications that could be potentially attractive for acceleration.

Accelerated data structures and HW/SW integration

The author has found little work which has attempted to address high-level applications that are composed, largely, of data-intensive kernels relying on data structures. Moreover, to the author's best knowledge there has been little that considers the interplay of hardware and software between high-level and abstract languages and hardware accelerators in a generic way.

This thesis proposes, using the Java programming language as a model, an approach to solve this using a generic data structure interface, run-time support to automatically select between hardware and software implementations, and a hardware priority queue implementation as a case study.

Polyhedral-assisted accelerator generation

I believe that polyhedral compiler technology can enable the generation of higher quality hardware accelerators given roughly unmodified input codes. Human intervention is still required to provide *scattering functions* that describe the desired behaviour of the accelerator and expose parallelism. This is in direct contrast to tools that require annotations about the nature of the hardware architecture. The selection of 'optimal' scattering functions is an active but non-trivial research area (see [20, 79, 80] for instance) and shall not be explored in this work.

The polyhedral model can be applied to the application code which, given the legality of the scattering function, will produce a transformed polyhedral representation of the target code. I introduce a back-end phase, to generate the accelerator design in the form of synthesisable HDL code. The basis for the accelerator is a newly proposed target architecture which is amenable to polyhedral mapping. Finally, existing HDL synthesis tools are used to produce executable FPGA configurations. Software code to control and manage the accelerator from the host system can also be automatically generated. There is the obvious interplay between the results from synthesis and the scattering function selection, which provides easy and usable design-space exploration.

3 HARDWARE DATA STRUCTURES

It is universally accepted that FPGAs can be used as accelerators to rapidly perform intensive computational work, offloaded from a co-operating application running on a host system. However, it is generally assumed that the target applications are scientific and high-performance computing which revolve around numeric computation. In this chapter I look at how the promise of reconfigurable computing can be applied in a more generic sense, to applications that are not dependent on numerical kernels.

The next section introduces the case study that was explored as the basis for research of this topic—graph applications and algorithms, and the priority queue data structure which is commonly used in their implementation. This is then presented in greater detail with the exposition of the hardware implementation and, moreover, how it can be used transparently within the software domain (Sections 3.2 & 3.3). The scalability of the design of both the hardware and software is crucial and explored using hardware/software co-design in Section 3.4. The test platform and performance results are discussed and concluded with my views on the short and long-term prospects of this approach (Sections 3.5 & 3.6).

Non-trivial data structures are both popular and pivotal in contemporary software applications. They define a way to organise data in a computer system memory and define associated algorithms, operations, and methods for accessing and processing this data. They are used both for the abstraction and storage of application information and are utilised in the implementation of many classical computer science algorithms. In many software applications there is a heavy

reliance on data structures as it allows for more modular, clean, abstracted code and more efficient data processing and algorithm implementation.

This motivates further investigation into whether reconfigurable hardware can be used to accelerate applications or parts of them that are reliant on high-level data structures. I start from the hypothesis that reconfigurable hardware can be used to implement data structures and associated algorithms and, in some cases, lend a desirable performance advantage compared to a software implementation. The target is to be able to provide tangible benefits to contemporary application developers that are utilising an object-oriented methodology, particularly based on the aforementioned data structures and algorithms. In this context, there has been little prior research conducted in this area.

3.1 GRAPH PROCESSING AND THE PRIORITY QUEUE

Graphs are a mathematical representation of a set of objects connected by links. Each link represents a relationship between the two connected objects, and can be attributed with properties that describe this relationship. This construct is prevalent in many areas of computing, particularly for modelling networks and relationships. For example geographical topology, the hierarchy of a computer website, or relationships within a social network can all be modelled in this way. The data embedded within the graph can be analysed by traversal. Computer algorithms and applications have been developed to facilitate with this process of data analysis, and graphs have now become a fundamental computing structure.

Graph traversal involves visiting the vertices within a graph, often by following the edge relationships.

3.1.1 *Graph theory*

A graph is an ordered pair: $G = (\mathbf{V}, \mathbf{E})$. $\mathbf{V}$ is a finite set of vertices, representing the collection of objects that are modelled. $\mathbf{E}$ is a finite set of edges, where each edge itself is a pair (u, v) with $u, v \in \mathbf{V}$, that defines the connection between the two vertices u and v.

In an undirected graph, $\mathbf{E}$ is composed of unordered pairs such that the edges (u, v) and (v, u) are equivalent. Therefore, there is no notion of the direction of an edge because they are treated symmetrically. On the other hand, in a directed graph $\mathbf{E}$ is composed of tuples

where (u, v) signifies an explicit directional connection from vertex u to vertex v. Graphs can be drawn pictorially with vertices as points with lines or arcs between them to represent the edges. Moreover, arrowheads are employed on directed graphs to visualise the direction of an edge.

Edges can be annotated with attributes, for example a weighting $w(u, v)$. A weight value may represent a distance, capacity, or a more general cost associated with the edge relationship between the two connected vertices. A weighted graph is a common graph variant where all edges have a weight.

A deeper treatment of graph theory can be found in texts such as [34, 42, 86].

The internal computer representation of graphs are typically either matrix-based or list-based structures. Matrix-based representations have been used in the past for performance or to gain greater low-level control of implementations and allow for more compact storage of dense graphs. In contrast, due to the higher level of abstraction, list-based structures make more sense logically to modern developers and are easier to work with due to provided flexibility to add and delete nodes, particularly in contemporary object-oriented (OO) languages like Java. Moreover, they are more memory-efficient for sparse graphs. For these reasons, the list-based representation has strong universal appeal and is the focus of this work.

3.1.2 *Prim's algorithm for computing Minimum Spanning Tree*

A common graph operation is the computation of the minimum spanning tree (MST). Provided with an input graph formed by a set of vertices and a set of weighted edges, the weighted MST is the subgraph which contains all vertices connected by a sub-set of edges of which the total edge weight is equal to or less than that of every other possible spanning tree. An example of a weighted MST is shown in Figure 7. This means that a MST will always have a minimum total edge weight, but there may be more than one unique tree that meet this condition.

A, sequential, pseudo-code implementation of Prim's algorithm for an adjacency-list graph representation is shown in Algorithm 3.1, ad-

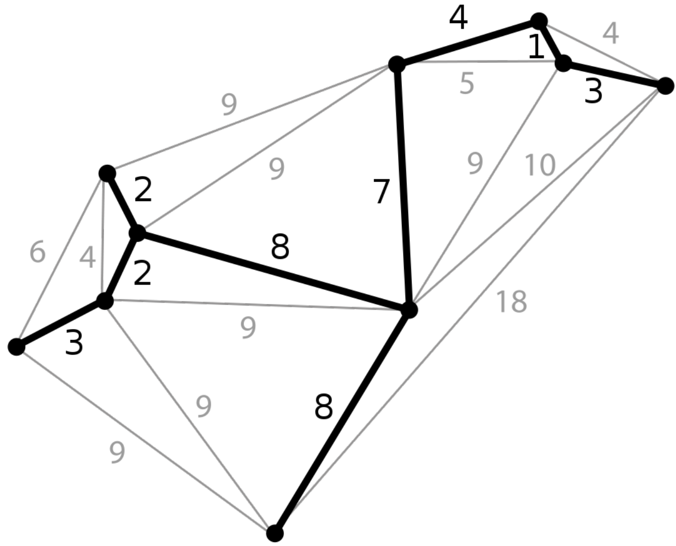


Figure 7: Arbitrary graph with the minimum spanning tree sub-graph highlighted with thick-lined edges.

apted from [42, 69]. It starts with an initialisation phase (L2–10) that configures the local data structures (a dictionary to keep track of visited vertices and a list of edges of the minimum spanning tree). Moreover, the Q variable is used to represent a list of items that is indexed by weight, or otherwise sorted, such that the lowest weight element can be extracted. A start vertex is chosen randomly and all incident edges are added to Q. The main loop of the algorithm (L12–23) iterates over the edges in weight order, ignoring it if the target vertex has already been marked. Otherwise the target vertex is marked and all incident edges with unvisited target vertices are added into the queue for further processing.

Prim's algorithm can be considered to be *greedy*, as at each main iteration the locally optimum lowest-weight edge is selected into the minimum spanning tree. A practical implementation of this algorithm would rely on a priority queue (PQ) data structure to instantiate the Q variable. A priority queue behaves like a sorted queue where the highest priority element is returned on an Extract operation.

It can be seen that the performance of the algorithm is dependent on the algorithmic complexity of the PQ. A naïve implementation may use a linear algorithm to perform the Extract and the Insert operation resulting in $O(V)$ algorithmic complexity. However, assuming a more advanced implementation based on a heap, Extract and In-

sert take $O(\log V)$ time, and then the algorithm can be executed in $O(E \log V)$ time [34].

The MST algorithm is interesting in the context of this research as the above formulation uses a priority queue which can be efficient on a single processor system. Parallel implementations often favour adjacency-matrix representation and search entire arrays every iteration step instead of keeping them in order which is less efficient for non-dense graphs but easier to implement concurrently. Parallel implementations typically have a computation time of $O(\frac{V^2}{p})$, where p represents the number of processors used, excluding overhead due to communication [45]. Instead of using the FPGA to implement an application-specific concurrent version of the MST algorithm, this research differentiates itself by exploring the potential of attacking the priority queue data structure in a more general way.

3.1.3 *Software priority queue implementation*

Efficient execution of Prim's algorithm is heavily dependent on the underlying priority queue implementation that has been selected for use. The algorithmic complexity of common software PQ implementations is shown in Table 4. A linked list implementation does not have very attractive performance characteristics but it is relatively trivial to implement. A binary heap is more complex but still a common data structure and has very desirable efficiency improvements. Standard utility libraries such as java.util.PriorityQueue are based on a binary heap implementation. The Fibonacci heap is the most efficient implementation but comes at a large implementation complexity and development effort.

Table 4: Algorithmic complexity of selected software priority queues [105].

IMPLEMENTATION	COMPLEXITY	EXTRACT-MIN	INSERT
Sorted linked list	worst-case	O(1)	O(n)
Unsorted linked list	worst-case	O(n)	O(1)
Binary heap	worst-case	O(log n)	O(log n)
Fibonacci heap	amortised	O(log n)	O(1)

3.2 HARDWARE PRIORITY QUEUE

The concept of a priority queue is no stranger within the hardware domain and there have been numerous publications on efficient implementations [18, 74]. But this work has typically been focused on their use within the context of communications and computer networking hardware appliances, for instance, prioritised packet queues for network routing. This difference in objective has, naturally, led to different design decisions being taken.

A binary tree of comparators is one such implementation architecture that has been investigated. In this approach the queue elements are fed into a comparator tree with the highest priority element propagating to the top and out of the tree. At a glance this architecture is conceptually simple and thus attractive to implement. However, there are some drawbacks in regard to scalability and performance, as propagation delays through the comparator network can significantly impact the operating frequency of the circuit. Variations of this approach have included pipelining to increase achievable clock frequencies but at the cost of increased latency. Further work has explored reusing the comparator network to service multiple queues and thus potentially hiding the pipeline latency [83].

For cases where the number of priorities is relatively small, a 'bucket' like approach has been proposed. There is a buffer for each available priority choice–whether physically or logically implemented–that maintains ordering for that priority [30]. Again, this approach has scalability drawbacks when used in conjunction with more generalised applications, like Prim's MST algorithm, because of the limitations on available priority values.

A pipelined heap structure is proposed in [18]. It has a nice scalability property since it leverages embedded RAM for the primary storage of the actual queue data. The actual binary-heap operations are, relatively, complex to implement; this work has taken care to separate the operations in a way that in amenable to pipelining. Pipelining significantly improves the performance to constant time complexity, for multiple accesses. However, there is still a latency hurdle for single operations.

The shift-register approach constructs a priority queue from a set of special storage elements that incorporate a comparator and decision logic into a shift-register [74]. The shift-register is arranged such that

one-end is the highest-priority and all blocks are sorted such that the other-end is the lowest priority. The highest-priority end enables constant time insertion and extraction. Each new queue entry is broadcast on a data bus read by each block which compares this new value with the value it is currently storing. The blocks broadcast the result of the comparison to those either side of it and use this information to make a decision on whether its storage element value should be modified. This approach is not complex and has attractive performance characteristics. Resource demands and the necessity of a long (and, typically, wide) bus connected to each block poses a limit to the potential scalability of the architecture if very large queues are desired.

Systolic array approaches have been proposed to address some of the issues with shift-registers [74]. This architecture eliminates the broadcast bus for new entries by making the contents of an entry element be propagated through the chain of blocks until it reaches the correct location. This eliminates the bus loading issue but comes at the cost of increased resources as an extra storage register will be necessary in each block to contain the propagating element. These changes also mean that the PQ will not be fully sorted until potentially many cycles later (linear time complexity), although the constant time insert and extract attribute can be maintained. Hybrid architectures that combine the benefits of the systolic and shift-register approaches have been proposed recently which could be attractive for very large queues, at the cost of added complexity [74].

I believe that for this work the shift-register is a very attractive implementation architecture. It is not overly complex, provides attractive performance characteristics, and is suitable for use within general purpose computing applications. It is easy to scale, from a development perspective, provided the queue size does not become too large, where the internal bus becomes a burden or otherwise the device capacity is reached. Moreover, in my proposed reconfigurable computing environment we need not limit the PQ to hardware only, therefore, software techniques can be applied to manage the queue growth as revealed in Section 3.4.1.

3.2.1 *Hardware PQ top-level architecture*

In this thesis I have implemented a hardware PQ using the shift-register method, as found in the literature. The top-level architecture of the priority queue is depicted in Figure 8. In this diagram it can be seen that the PQ is constructed from a number of shift-block storage elements that contain each element of the queue. Connected together, these shift-blocks form a shift-register chain that is the basis for the priority queue. New elements that enter the queue are broadcast along a main bus (new_entry_bus) that spans the entirety of the queue and is connected to each shift-block. This connection is made as each shift-block needs to make a comparison with the new entry value and the currently stored value within in. The shift-blocks are linked with control signals that serve to communicate the results of these data comparisons with the directly neighbouring elements. The entire architecture has been described using VHDL and utilises `generic` parameters and `generate` statements which facilitates trivial scalability of key parameters—including data and priority width, and queue size—at synthesis time.

3.2.2 *Shift-block storage element architecture*

Figure 9 depicts the storage block used within the shift-register chain. Every clock cycle the contents of the storage element is updated to either: a new value broadcast on new_entry_bus, the value stored within directly neighbouring left or right shift-blocks, or it maintains its current value. This decision is determined locally in the control unit using the decision matrix shown in Table 5, which assumes the shift-register is implemented left-to-right with the right-most element containing the highest priority element (and that the primary PQ operation is Extract-Min; that is, the highest priority element has the lowest weight value).

The `comp_out` signal is generated locally within each shift block and is asserted if and only if the weight value on the `new_entry_bus` is less than the currently stored weight value. This signal then propagates to the adjacent left shift-block as the `comp_in` input signal. For an Insert operation this chain of comparisons will be performed and, working from right-to-left (low-to-high weights), the comparator output remains low until the first element is found which is greater than the

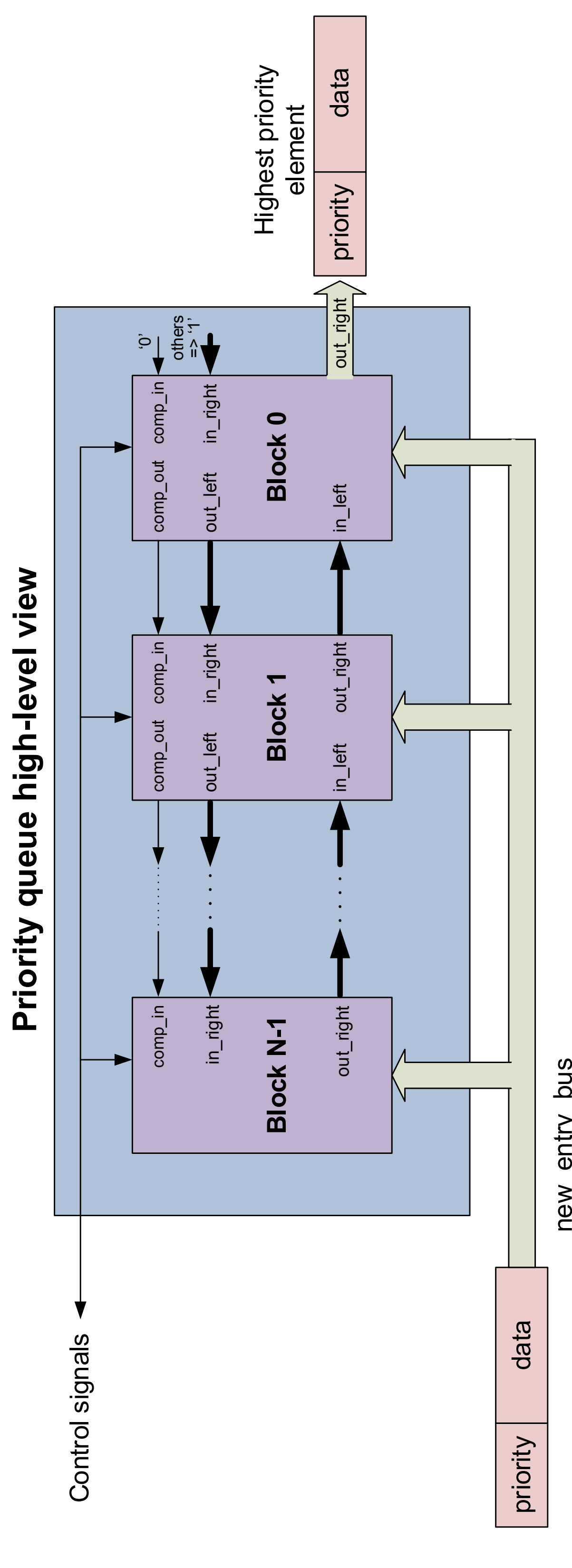


Figure 8: Full shift-register architecture priority queue implementation showing the multiple storage blocks and their communication pathways.

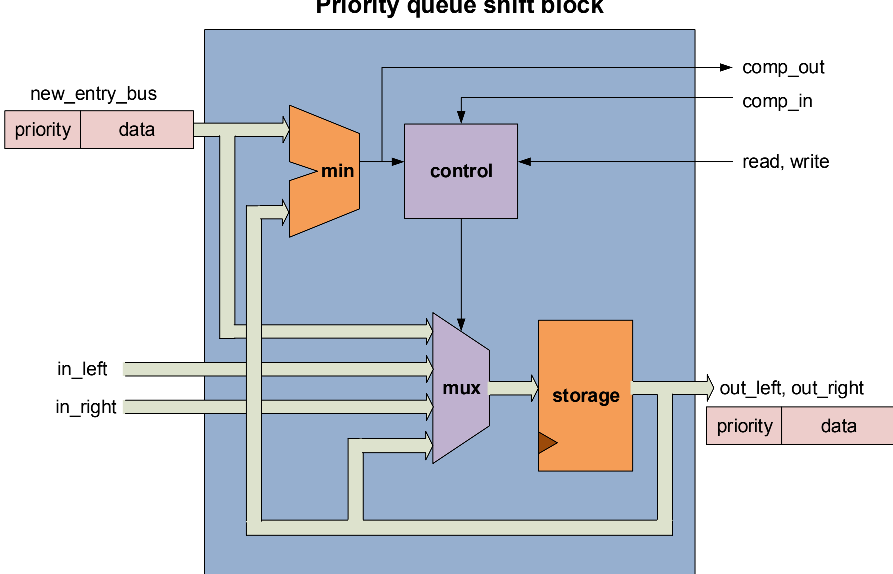


Figure 9: Storage element used in the construction of the shift-register priority queue.

new_entry_bus. This element will then replace its storage with this new value. Following that, all the remaining left-most shift-blocks will have both comp_in and comp_out asserted and so all blocks will update to the value on the right, therefore, shifting the entire queue along one space to the left.

Extract is trivial in comparison since all shift-blocks read the in_left value, therefore, shifting the entire queue one space to the right, with the highest-priority element exiting the queue.

In this implementation, the priority values are treated as integers stored within bit arrays. This results in the synthesis of a simple comparator. The width of the priority bit array is defined as a generic parameter and can be scaled easily to allow for more or less unique priorities, as desired. The actual data values are also treated as bit arrays which can be scaled in width.

3.2.3 *Performance characteristics*

The design of the hardware PQ has an algorithmic complexity of $O(1)$ for both the Insert and Extract operations. Both of these operations take only a single clock cycle to complete at the hardware level, although it will naturally take longer when accounting for the communications overhead to and from the processor on the host system to

Table 5: Local decision matrix to determine storage block operation each clock cycle.

OPERATION	COMP_OUT	COMP_IN	STORAGE INPUT
Extract	X	X	`in_left`
Insert	0	0	*maintain value*
Insert	0	1	`in_right`
Insert	1	0	`new_entry_bus`
Insert	1	1	`in_right`
X	X	X	*maintain value*

actually undertake these operations. This demonstrates excellent theoretical performance; better than the common software approaches (shown in Table 4).

While the performance in terms of clock cycles is $O(1)$, it should be considered that the length of a clock cycle—or its reciprocal, the frequency—is influenced by the length of the PQ. Theoretically, the propagation delay on the bus has a logarithmic relation to the size of the PQ when routed in a tree like structure, because the electrical length increases by level and not linearly. However, as will be seen in the experimental results of Section 3.5, the limiting factor for the scalability of the PQ is the size of the FPGA device, in terms of available logic elements, and not the propagation delay on the bus. Therefore, for a given device we can consider that the hardware PQ does exhibit $O(1)$ performance.

3.3 HARDWARE/SOFTWARE INTERFACE

This work has been based on Java as the language for application development. It represents a modern, object-oriented language that is commonly used to implement mobile, desktop, and enterprise applications. The success of reconfigurable computing hinges on the ability to easily and transparently incorporate hardware acceleration with the application domain. In this research I propose the use of a layered software stack that serves as the communication mechanism between the application and the underlying FPGA hardware. Using Java as the case study, such an approach can be implemented using native interface calls via the Java Native Interface (JNI) mechanism or

In some experiments a J2ME KVM runtime was used that provided a KNI interface, as an alternative to JNI.

similar depending on the deployment platform. The interested reader is referred to [67] for more thorough coverage of the JNI specification.

Figure 10: Software layers between Java application and physical hardware

3.3.1 *Java/Hardware interface implementation*

The communication hierarchy is depicted in Figure 10. Working bottom-up from the physical layer, a Native Interface (in the case of Java) driver was developed that implements the required platform specific functionality to facilitate communication with the FPGA device and, moreover, supports the protocols employed by the application-specific hardware modules.

The HardwarePQ object is defined in Java as containing native methods: the methods are declared in prototype form but the definitions are not present. Instead the method definitions are in a native system library compiled from C source code and linked in to the JVM at run-time. The native methods implement functions to initialise the FPGA via low-level system/device methods which are platform specific. Then data can be written to and read from the defined FPGA system registers, using the Java objects to transparently perform the necessary communication transactions with the FPGA. In the case of the priority queue rather than passing raw data values to and from the hardware, object references are used at the native level and thus the actual objects need only be manipulated within the application level.

Java native methods actually target the system itself (hardware, OS, and libraries) and, therefore, are commonly specified in C, C++, or assembly.

The write process, as a Java native method implemented in C, follows:

```
fpga_info* fpga_regs = [initialisation of FPGA registers]
void JNICALL Java_ns_writeObject(jobject obj, jint weight) {
    long objPtr;
    objPtr = ((long) env->NewWeakGlobalRef(obj));
    // write weight
    fpga_regs->pq.weight = weight;
    // write object pointer
    fpga_regs->pq.data = objPtr;
}
```

It is quite a straightforward process that uses the `NewWeakGlobalRef` method to retrieve a pointer to the current Java 'object' that we want to store in the queue. The special `fpga_regs` variable is a pointer to an FPGA data structure that is mapped to the low-level FPGA memory/register space on the device. Prior to making any hardware calls this must be initialised, which can be performed when the PQ is instantiated. Extracting and reading an item from the PQ is even simpler:

```
jobject JNICALL Java_ns_readObject() {
    // read object pointer
    long objPtr; = fpga_regs->pq.data;
    return (jobject) objPtr;
}
```

Because the object pointer is stored directly, there is no need to perform any further manipulation. It can be casted as a `jobject` type and will be correctly recognised by the runtime virtual machine.

3.3.2 *Transparent application usage*

The developed HardwarePQ library utilises the native methods to interact with the embedded hardware priority queue. This class implements a generic PriorityQueue interface, much like a software implementation would, so that the same PriorityQueue methods are exposed at the application level whether using the software PQ or the hardware version. This enables seamless use within new applications with very minor code changes required to existing ones. As contemporary applications are often developed according to interfaces rather than implementations, utilising the hardware PQ is as simple as replacing the object instantiation with:

```
PriorityQueue pq;
if (FPGA.isPresent()) {
    pq = new HardwarePQ();
} else {
    pq = new BinaryHeapPQ();
}
```

This instantiation represents an employment of the Abstract Factory design pattern, a common template for providing run-time selectable encapsulation in software applications. I believe this approach

is extremely accessible to application developers and allows them to easily target both cases, using only the hardware acceleration if its supported by the underlying computer system.

3.4 DESIGN FOR SCALABILITY

In this section I show how a hybrid hardware/software PQ can be extended and scaled to incorporate additional functionality that may be necessary in various applications. Three common application use-cases have been considered: increasing queue length, dealing with priority values beyond simple fixed integers, and implementing additional operations.

3.4.1 *Hybrid HW/SW queue for length extension*

In real-world applications, it is of significant importance to be able to scale the PQ in maximum length, necessary to handle variable and possibly large data sets as input. I have already discussed how the hardware can be adjusted, in Section 3.2.1. The hardware queue can be scaled to a relatively 'large' size but comes at a resource cost and is prone to be affected by issues such as bus loading and routing that impacts potential clock frequency and hence performance. However, regardless of the architecture and design, ultimately the length of the hardware queue is constrained by the physical resource limitations of the FPGA device. Thus gracefully handling the situation where there is not enough hardware is more necessity than nicety.

The software PQ will be constrained by available system memory which is expected to be orders of magnitude larger than what is achievable with the proposed hardware PQ alone.

To address this limitation, I have applied a hardware/software co-design paradigm to the problem. The proposed implementation logically extends the hardware PQ by appending a software binary heap PQ to the lower-priority end. The binary heap implementation was chosen as it is efficient, robust, and already available within standard libraries, for instance `java.util.PriorityQueue`. The realisation of this is a high-performance hardware PQ to maintain the high-priority end of the data structure and a low-cost, scalable, software memory system to maintain the low-priority segment of the data structure, that is virtually unconstrained in length.

When the hardware queue becomes full, excess low-priority elements are moved into the software queue. Likewise, as the hardware queue empties out, the highest priority elements within the software

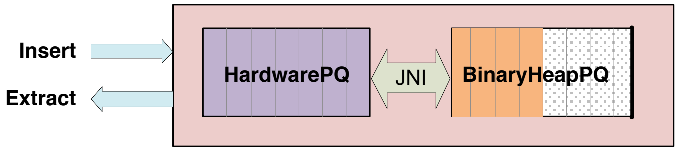


Figure 11: Logically extended HybridPQ utilising the Hardware and Software queues internally.

queue can be removed and inserted into the hardware queue. This simple logic can be encapsulated into a new class that is dubbed the HybridPriorityQueue. This still implements the PriorityQueue interface for ease-of-use, and therefore requires minimal application modification, and is constructed using internal instances of the separate hardware and software components, depicted in Figure 11.

The actual hybrid control algorithm that describes the mechanics of the hybrid operation is shown in Algorithm 3.2. The Insert and Extract-Min procedures are shown, and they are simple and straightforward. The former handles the special case where the hardware PQ is full by comparing the priority of the inserted element with the current lowest-priority (maximum value) within the hardware PQ, i.e., its last value. If the inserted element is greater than this maximum in the hardware queue then it is known that surely this element need not be inserted into the hardware PQ. Otherwise, the element is higher priority than the last element in the hardware PQ and so must be inserted into the hardware PQ. However, since the hardware is full, we must make a copy of the lowest-priority element within the hardware PQ and insert it into the software PQ. Then we can force an insert of the new element into the hardware PQ, knowing that the extra element will automatically be expunged on the next shift cycle. Extraction is always performed from the hardware PQ, so if the software PQ is active then a value must be popped and inserted back into the hardware PQ to occupy the just freed space.

Peek-Max is a newly proposed hardware operation to enable the Insert procedure to work correctly. In practice, it is trivial to implement by connecting the lowest-priority element to the register-file interface so it can be read.

With this approach, the algorithmic complexity is modified—when the hardware queue is not full it is utilised for all operations and thus the $O(1)$ complexity is maintained, but when it is full the software PQ is used in addition to the hardware PQ making the worst-case com-

plexity $O(\log n)$ for all operations, as the number of stored elements reduces and the software PQ becomes empty, then again the pure hardware PQ performance will apply. This demonstrates that we can meet hardware constraints while still maintaining good performance (algorithmic complexity characteristics), while also scaling the architecture for problems requiring very large queue sizes.

3.4.2 *Priority range mapping*

The proposed implementation restricts element priority values to integers (represented internally as bit arrays). Integers have the nice property that hardware comparators are relatively cheap thus suiting a hardware implementation that is already resource-constrained. But for real-world applicability the queue must be useful for applications that instead require different priority sets, for example, floating point numbers or alphanumeric strings.

However, again, we can leverage the principles of hardware/software co-design and incorporate functionality into the HybridPQ that can convert priorities between the application priority set and the internal PQ priority set. For example, assuming a priority set of 0.0 to 1.0 the latter can be mapped to the highest priority value, 0.5 the middle priority value, and so on. This approach maintains the hardware simplicity of the proposed hardware PQ yet facilitates the usage of complex priority sets thanks to domain mapping easily achieved on a per-application basis. For a given priority data width the resolution is fixed so this needs to be considered when performing the mapping to ensure enough distinct priority values are available, however, this can be scaled during hardware synthesis.

3.4.3 *Extending functionality in the software domain*

Decrease-Key is an additional priority queue operation which updates the priority value of an element that is already in the queue, thus promoting its importance, remembering that elements are accessed in ascending priority order (thus, decreasing a priority value means it could be accessed sooner). Although, classically, this is not a required priority queue operation it is important in graph processing. For instance, it is used in Dijkstra's algorithm for shortest path calculation. A limitation with the proposed hardware PQ architecture is that this

operation is not directly supported in hardware. Hardware modifications could be made to implement this, it would be at a cost of hardware resources and possible performance potential.

Instead, Decrease-Key can be easily emulated in software. Rather than search the PQ for the element and update it, another entry for the element can be added into the PQ with the updated priority weight. Naturally the PQ will maintain the order such that the newly updated element will be retrieved before the older *stale* instance. Software facilitates this process by ignoring the stale element when it is finally retrieved by updating an internal map or dictionary structure, at a small performance cost.

3.5 Performance results

In this section I demonstrate the performance of the hybrid hardware PQ implementation on an embedded systems platform. Given the substantially weaker CPU strength of typical embedded systems, compared to workstations, they are an attractive use case for this type of acceleration. The experiments test the PQ across a range of conditions by scaling the input problem size.

3.5.1 *Implementation environment*

The experiments were based on an Altera Nios II SoPC reconfigurable computing environment. Nios II is a soft-processor developed by Altera for use on its FPGA products. The SoPC Builder tool provided with this processor allows for complete customisation of the computing environment by specifying the processor features and specifications, including support for custom instructions, selection of system components including the number and type of memories, and the inclusion of any other custom hardware models. After specification it automatically generates the appropriate high-level system description by taking care of all communication networks, bus arbitration, and prepares the software environment by combining necessary device drivers and generating an accurate memory mapped system description. I have chosen this platform as the test-bed for research with the Hybrid PQ because it is a robust and well supported tool chain that facilitates the rapid exploration of both software and hardware approaches.

In Figure 12, it can be seen that the internal system topology is representative of other more traditional reconfigurable computing environments, for example [108], with a standard system bus interconnection network that connects processors and memories. Albeit, because it is a soft-processor environment both the processor and additional hardware components—the Hybrid PQ—are instantiated within the same device. Moreover, this approach is portable to other platforms and does not depend upon any specific platform features.

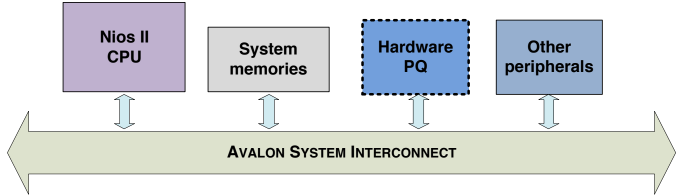


Figure 12: System architecture of the Hybrid PQ reconfigurable computing test environment.

On the Nios II processor, I used the the Java 2 Micro Edition (J2ME) virtual machine to run the application code. J2ME can be seen as a subset of the standard or enterprise Java platforms and thus the application code remains portable between the Nios II test system and any other typical Java environment. Low-level system calls to the hardware PQ have been implemented using C and are included in the Java application code by means of the J2ME Kernel Native Interface (KNI) mechanism. These system calls are specific to the platform but as they only expose methods for physical communications, with protocol and functionality implemented in the Java layer, the functions are both simple and easily ported.

3.5.2 *Test parameters and synthesis results*

In this next subsection I present my results running a Java minimum spanning tree computation on J2ME on Nios II. We shall compare the performance between running with a standard software PQ and with the proposed hybrid PQ. In both cases system clock frequency has been set at 100 MHz for both the Nios II processor and the hardware PQ. The tests were performed on a Terasic DE2-70 development board which features an Altera Cyclone II EP2C70 FPGA device. Using this platform I tested with a hardware PQ with 512 elements of 32-bit data width and 16-bit priority width. Priority values were treated as

integers and the data values were actually 32-bit references of the Java objects.

In Table 6, the synthesis results for the hardware PQ are shown. It can be seen that a queue length of over 1000 elements at moderate-high frequencies are clearly achievable in current-generation mid-range FPGA devices. However, it is clear that as the length grows larger the resource consumption increases linearly, and the maximum clock frequency reduces logarithmically due to longer signal travel times. This is caused in large part due to the inherent ineffectiveness of the chosen shift-register architecture as a result of the propagation delays in the complex routing network between the register elements. But in saying that, it is evident that the physical device size is the most immediate synthesis constraint which reaffirms this architecture is a reasonable compromise for queues in this size range.

Table 6: Synthesis results of the hardware PQ for Altera EP3SE260C2 device at selected queue lengths.

LENGTH	128	256	512	1024	2048
ALUTs	8540	17069	34094	67853	137390
Registers	6547	12947	25751	51347	102545
Utilisation	4%	8%	17%	33%	68%
F_{max} (MHz)	291.72	266.24	239.81	203.09	150.60

3.5.3 *Performance comparison*

Overall, I have found the performance of the hybrid hardware/software PQ to be very promising across the experiments, computing Prim's minimum spanning tree algorithm (Algorithm 3.1). Three graph types were tested: random graphs with a fixed number of vertices and an edge density representing the average number of edges per vertex, 2D-grid graphs with a fixed number of vertices and edges, and 3D-mesh graphs with a fixed number of vertices and edges. Apart from very small graphs, for example a 2D-grid with only 9 vertices and 12 edges, the hybrid PQ has consistently performed faster or as fast as a pure software PQ implementation. The actual time taken to complete this calculation ranges for 0.05 s for the computation of a 50 edge random graph through to over 5 s for random 3D-mesh with a dimension

of 9 (or 729 vertices). Thus the MST calculation is compute intensive and the attained speedup is significant. Exemplary performance results are illustrated in Figure 13.

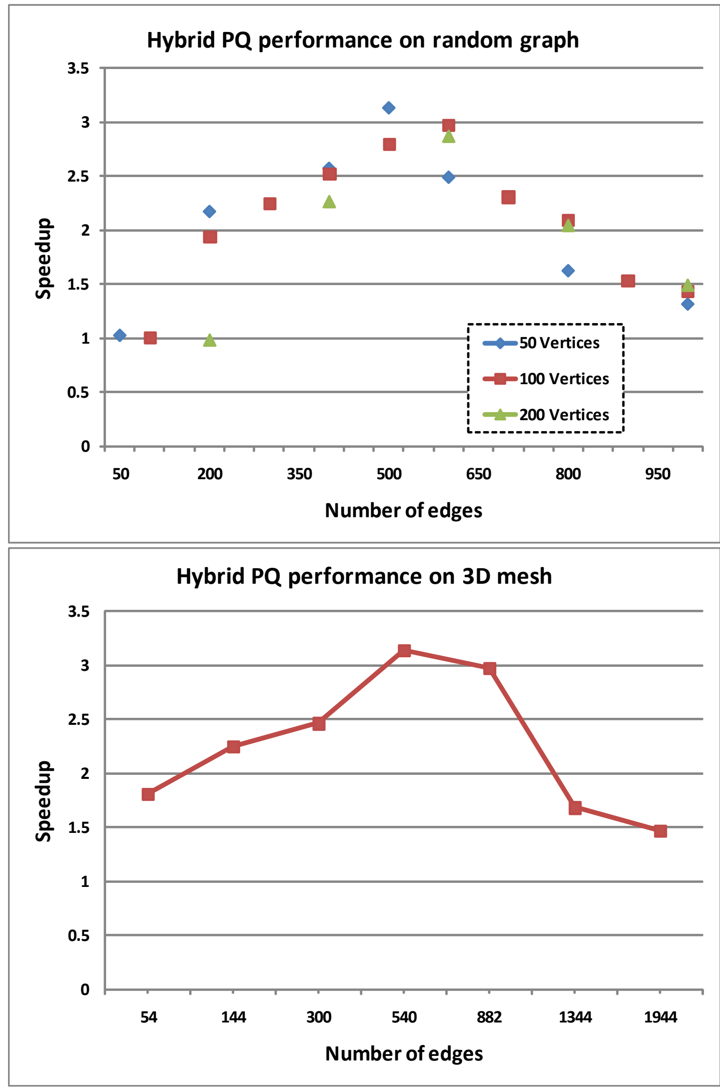


Figure 13: Comparison of MST computation when using pure-software binary heap PQ implementation and the hybrid hardware/software PQ with a HW size of 512 elements. Top: random graph data set, bottom: random 3D-mesh.

As the size of the graph increases—in either number of vertices (and hence edges), edge density, or both—the performance speedup from the hybrid PQ increases as the additional hardware communications overhead is offset by the faster responsiveness, as compared to operations using larger software binary heaps. Peak performance is achieved at a point where the hardware PQ capacity is fully utilised but the queue size does not extend considerably into the logically

extended software PQ portion and results in a speedup of approximately 3x over a pure-software implementation. This peak performance point is achieved at different graph sizes depending on the graph structure, for example, for the random graph case it occurs when the number of edges are approximately the same as the size of the PQ but for 3D meshes the number of edges is approximately twice the size of the PQ.

When the input graph size is considerably larger than the capacity of the hardware PQ the performance drops off but is not significantly degraded as compared to the pure-software case. This can probably be attributed to the fact that the hardware PQ is typically used enough in the computation as the PQ is either being filled up or emptied that it again offsets the added communication overhead necessary with the logically extended hybrid queue approach. Moreover, this shows that the hybrid PQ is robust enough to be used in a wide-range of general-purpose applications where the data size is not known and performance will at least match that of a software implementation.

3.5.4 *Embedded vs workstation platform*

In this research I have proposed a generalised approach to integrating hardware data structures within contemporary application development using the priority queue as a case study. However, the test platform is more akin to that of an embedded systems use-case with resource constraints, a relatively slow processor, and the benefit of tight system integration. When the same approach is applied to workstation-level reconfigurable systems, for example the XD1000 [108], although the overall system topology is similar there is a significant performance bias toward the general purpose processor. In these systems the processor is more powerful and at least an order of magnitude faster than the Nios II soft-processor that was tested.

Even when using fast and expensive FPGA devices it is completely unrealistic to expect the performance of the hardware PQ to scale in the same way. Moreover, the communication overhead between the CPU and FPGA is greater due to the looser system integration. For these reasons, at the present time I do not expect to see a performance benefit when using the hybrid PQ approach with these workstation reconfigurable computing systems. This could possibly change in the

future as FPGA technology advances and the communication barriers are reduced.

Preliminary results were conducted on a XD1000 system which confirmed these findings. The FPGA was positioned in a CPU socket and connected via HyperTransport to the host CPU. A standard desktop-edition Java stack was used, with only minor differences to port the JNI methods to this environment. The hybrid PQ was no faster than using software only, primarily because the latency and overhead with the implementation was masked by the clock frequency advantage of the CPU.

3.6 CONCLUSIONS AND FUTURE DIRECTIONS

Data structures are commonly used abstractions in contemporary object-oriented applications. Using the priority queue as a case study, an up to 3x application performance improvement has been realised over an efficient pure-software implementation using the test platform. Ultimately at the current state of FPGAs and reconfigurable systems architecture, this approach may not have the raw performance to compete with a pure-software implementation on workstation-class hardware. But I have shown that there is merit to undertaking further research in this domain and other hardware-based data structures may show an even larger performance benefit. This direction is being actively explored within the PARC group at The University of Auckland.

From the perspective of application development, my approach to hardware/software integration is very transparent and gels well with contemporary object-oriented design methodologies by conforming to common interfaces. This has resulted in excellent accessibility for software developers to utilise hardware components in their design, and demonstrated an ability to potentially implement new functionality using hybrid hardware/software features. The recent thesis by Bloom [19], citing the thesis author's work [27], has further explored this pioneering direction and presented an OS-based approach to handle overflow in hybrid data structures.

The successful implementation of a hardware priority queue holds promise for the development of a general-purpose library of hardware-based data structure building blocks that can make a significant contribution to reconfigurable computing. It will allow for better utilisa-

tion of the reconfigurable platform and, coupled with the straightforward and transparent usage model, an attractive stepping stone to a more complete high-level language synthesis framework.

It can be concluded that the initial hypothesis is supported by this thesis findings. There certainly are some tangible benefits to data structure acceleration, particularly given the conditions where collection size is small, or the platform is relatively weak in GPP power. Given the holistic approach of this thesis, the remainder of the thesis concentrates on the acceleration of numerical codes, found in nested loop kernels.

Algorithm 3.1 Prim's algorithm to compute the minimum spanning tree of a undirected weighted graph.

Require: Graph G as adjacency list representation
Ensure: *mst* is a list of edges characterising the MST

```
1:  kernel MINIMUM-SPANNING-TREE(G)
2:      mst ← []
3:      for all v ∈ Vertices[G] do
4:          marked[v] ← NIL
5:      end for
6:      Q ← []
7:      s ← RANDOM-ITEM(Vertices[G])
8:      marked[s] ← TRUE
9:      for all t ∈ Adj[s] do
10:         INSERT(Q, w, (s, t))
11:     end for
12:     while Q ≠ ∅ do
13:         w, (s, t) ← EXTRACT-MIN(Q)
14:         if NOT marked[t] then
15:             marked[t] ← TRUE
16:             APPEND(mst, w, (s, t))
17:             for all t′ ∈ Adj[t] do
18:                 if NOT marked[t′] then
19:                     INSERT(Q, w, (t, t′))
20:                 end if
21:             end for
22:         end if
23:     end while
24: end kernel
```

Algorithm 3.2 Hybrid PQ business logic implemented within the software driver.

```
Require: hwpq and swpq as hardware and software PQ instances
 1: kernel InsertHybridPQ(p, e)
 2:     if hwpq is not full then
 3:         Insert(hwpq, p, e)
 4:     else if p ≥ Peek-Max(hwpq) then
 5:         Insert(swpq, p, e)
 6:     else
 7:         p′, evicted ← Peek-Max(hwpq)
 8:         Insert(swpq, p′, evicted)
 9:         Insert(hwpq, p, e)
10:     end if
11: end kernel

12: kernel ExtractHybridPQ
13:     p, e ← Extract-Min(hwpq)
14:     if swpq is not empty then
15:         p′, evicted ← Extract-Min(swpq)
16:         Insert(hwpq, p′, evicted)
17:     end if
18: end kernel
```

4 ACCELERATOR HARDWARE MODEL

This work specifically considers FPGA devices as accelerators. They are composed of—inherently undefined—reconfigurable logic; juxtaposed with some fixed-function digital signal processing (DSP) units, clock, and memory blocks. Designing a high-performance accelerator entails using the reconfigurable logic to create application-specific circuits, incorporating the fixed-function blocks as needed. The open-ended design-space provides an opportunity for clever and highly-optimised solutions in performance or resource use. But manual design remains a challenging proposition to meet performance goals while balancing engineering effort.

Mathematical calculations can be very demanding, particularly with floating-point numbers. DSP units found within FPGAs can accelerate certain critical operations, separate from other configurable logic.

This thesis proposes a convenient and intuitive architectural model that is well suited to the polyhedral approach. It provides a regular framework within which we can easily perform code-generation, yet still malleable enough to exploit the freedoms of FPGAs, which extends beyond any conventional parallel computer systems. This section describes how this architecture can be modelled and how it aligns to the overarching methodology for constructing a complete datapath.

Figure 14 depicts a high-level block diagram of a typical accelerator. Each accelerator shares the same modular framework: external data and control interfaces mated to the accelerator datapath of memory (M) and processing units (P) driven by a control unit.

4.1 FPGA PHYSICAL CHARACTERISTICS

Before discussing the target accelerator model, it is good to review the overarching physical characteristics of a typical, contemporary, FPGA

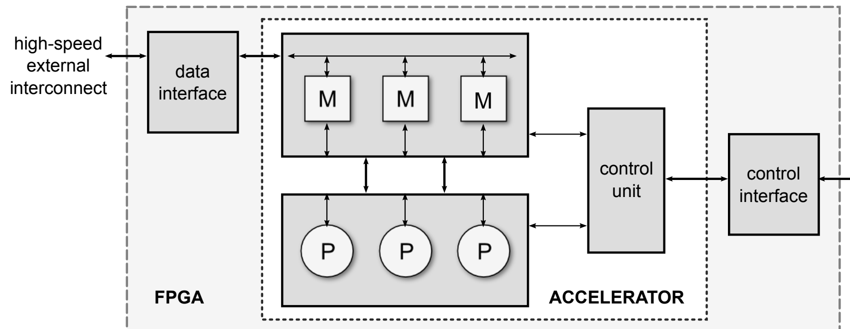


Figure 14: High-level accelerator architectural model. M is a memory and P a processing unit.

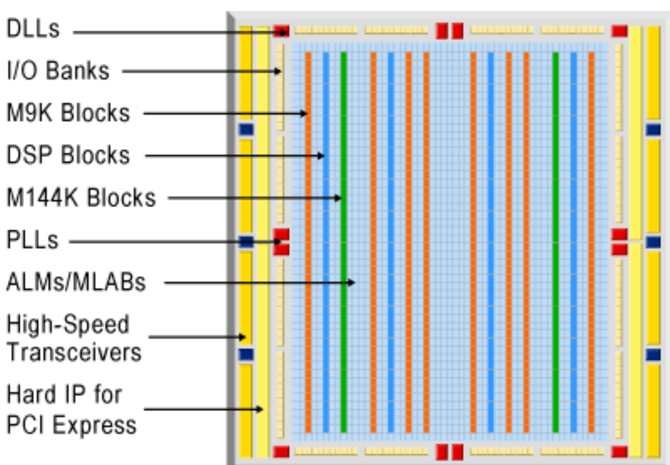


Figure 15: FPGA structural block diagram [8].

device. One such example, the Altera Stratix IV, is shown at a high-level in Figure 15, and served as the basis of the experimental work conducted as part of this thesis. The impact of the device characteristics should not be underestimated when developing any theoretical model or abstraction, and the architectural decisions are based on these findings.

4.1.1 *Resources*

While my treatment here is focused on the Altera Stratix FPGA family, the architecture of families from other vendors, e.g., Xilinx, is broadly similar at a high-level.

The majority of the device is, understandably, the core programmable logic fabric which is made up of small 'ALM' look-up table (LUT) modules. This itself consists of an 8-input fracturable LUT with two adders and two register cells. This module can be configured into seven different configurations: from a single 7-input LUT to two independent 4-input LUTs, but it is interesting to note that register cells

are directly embedded and physically spread throughout the device within the ALM.

Digital signal processing (DSP), or mathematical, modules are distributed across the span of the device with two to seven columns provided within the Stratix IV family of devices. Each module can be used to implement floating-point arithmetic functions much more efficiently than logic alone, and in this regard the tools support the generation of common floating-point arithmetic functions (in either single or double precision). These functions include: multiplication, division, multiply and add or accumulate, and dynamic shift functions. For many scientific codes, which are inherently computationally heavy, the availability of DSPs becomes the upper-bound on feasible accelerator size.

Memory resources are available in three tiers, categorised by the memory size (capacity) and available data ports (bandwidth). Generally, there is an abundance of available on-chip memory when compared with using registers as storage. The smallest of which, MLABs, are embedded throughout the device but limited in size to only 640 bits.They are useful for shift registers, FIFOs, and other small buffers. Larger M9K and M144K memory blocks are distributed across the device span and hold 9 Kib and 144 Kib of data, respectively, and both are true dual-port RAMs.

The M9K is particularly useful in accelerator applications as it offers an attractive blend between size and bandwidth (ports) which makes it useful for storing arrays (or rows of a matrix, for example). They are limited to a configuration size of 256x36 bits, however, multiple blocks can be combined together to facilitate larger requirements. This can be performed automatically by the synthesis tool-chain.

4.1.2 *Connectivity*

Resource elements are embedded within a large membrane of switched routing fabric. Within this there are many short-distance links and fewer fast global links which span large sections of the device. The amount of routing resources is carefully chosen by the vendor in such a way as to not limit the ability to connect any given design. At least, in this research work, I have found that routing resources has not been a limiting factor.

The maximum frequency of an accelerator is a function of the routing complexity so, ideally, designs should account for the placement and routing of resources across the device.

4.2 DESIGN OVERVIEW

The primary focus of my research is on the design of the accelerator datapath. I propose the datapath consists of only storage and compute elements embedded within an explicitly specified communications network. A storage element (SE) is an element containing data and a compute component refers to any processing element (PE). Two data flow constructions are ratified to support the desired polyhedral model:

1. computation, specifying an operation at instance i of a PE: $PE_i(in_1, \ldots, in_n)$. Inputs are references to SEs, specified in positional order within the parentheses. All PEs implicitly generate a single output.

2. assignment, which is used to specify data movement between two components: $sink \leftarrow source$. A sink must be an SE but a source input is a PE output or an SE.

These constructs are embedded within a control flow graph that describes the accelerator execution schedule with respect to two iterators, that directly correspond to a networking construction:

1. sequenced iteration, expressed as a for-type loop, results in a multiplexed connection between components, based on the selection variable (in most cases, t, time).

2. parallel iteration, expressed as a forall-type loop, results in a direct one-to-one connection between source and sink components of the data flow construction, independent of each other.

From this specification we can generate a complete datapath by ensuring each data flow construct has a corresponding hardware target. The network between components is deduced by analysing all inputs to computations and the right-hand side of assignment statements.

This final architecture resembles that of a Parallel RAM (PRAM) machine: an idealistic model with unbounded logically shared memory and unbounded processors [26, 92]. It is further classified by memory access; the proposed architecture supports a CRCW (Concurrent Read,

Concurrent Write) memory model for accesses, paired with a possible MIMD (Multiple Instruction, Multiple Data) execution paradigm. This is a significant proposition: the CRCW property stipulates, true, concurrent data operations, while MIMD stipulates that processing elements can perform different computations on different data. For traditional parallel computers such a model is unthinkable but it is tractable on the FPGA–made possible by creating many customised memories.

4.3 DATAPATH ARCHITECTURE

In this section I present a detailed look at components of the proposed architecture and the interplay between them.

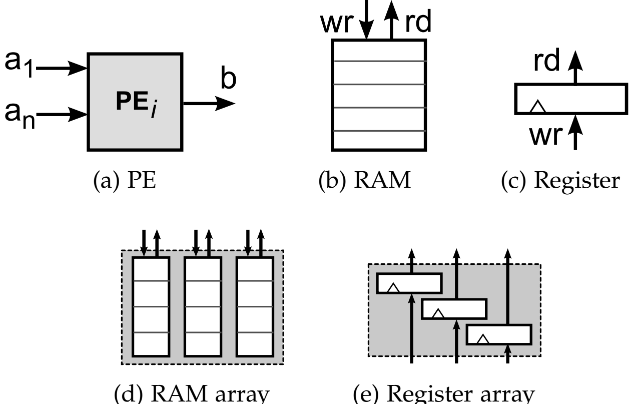


Figure 16: Breakdown of accelerator system components.

4.3.1 *Processing element (PE)*

An accelerator contains one or more *instances* of a processing element (PE) of a given type. A processing element can be treated as a, potentially compound, black-box execution unit. It can be considered capable of indivisibly executing an operation which maps to a single compound statement in a loop body. We can model a PE using a few basic properties: 1) n inputs $a_1, \ldots a_n$, 2) a single output b, and 3) computational latency, $t_{latency}$, that specifies the number of clock cycles before a valid result is produced (for PEs that support pipelined operation this is the pipeline period). This model creates a simple abstraction between algorithmic statements to the hardware domain.

If a loop body contains multiple statements, which are partial steps towards the generation of a single output value, they can be consolidated into a single larger compound statement prior to PE generation.

If an algorithm contains multiple unique statements then each is associated with a different PE type. Multiple PEs of different types can be instantiated in the datapath, creating the foundation for a MIMD model of execution.

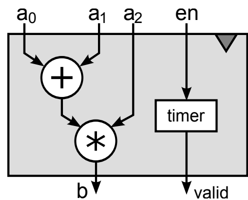


Figure 17: Datapath for a simple PE.

This work assumes all low-level mathematical functions are provided as reusable IP blocks, e.g. Altera FP megafunctions.

Figure 17 depicts a PE that implements a simple mathematical expression: $b = (a_0 + a_1) \cdot a_2$. The mapping between mathematical operators to a hardware unit is very straightforward, by leveraging external mathematical hardware blocks. Each operator block has a given latency; for example 14 cycles for double-precision addition, and 11 cycles for double-precision multiplication. The total latency of the computation ($t_{latency} = 25$ cycles, in this case) is embedded as a counter trigger within the timer block that is activated when the computational start signal (en) is asserted.

4.3.2 *RAM storage*

As described in Section 4.1.1, an FPGA device contains many, small-sized, embedded RAM blocks in several clusters, typically along the length of the device. These are feature-complete, dual-port, devices with full address control and single-cycle operation. They can be merged automatically, using vendor-provided libraries and tools, to emulate different size configurations. By leveraging many embedded memories the accelerator datapath can sustain higher-bandwidth and, therefore, higher-throughput operation, since each memory can be accessed within a single-cycle.

External memories (for example DDR2), on the other hand, have much greater latency and require more complex interfacing and data management in order to be well utilised within a high-throughput datapath. Managing these limitations for a performance objective will

ultimately require smart local caching techniques that also rely on the embedded device memories and registers anyway.

A single RAM provides relatively large and random storage within the architecture, but with limited concurrency. The dual-port interface to the storage contents limits data transactions to a maximum of two operations, on any location, per clock cycle[1]. Natively a RAM has a single dimension, storing all data as a linear array. This can be logically divided to provide multiple virtual dimensions to implement a 2D (or more) data space. Alternatively, given that the underlying FPGA fabric is embedded with many small RAMs, it can be advantageous to partition multi-dimensional data across multiple individual RAMs.

For example, a 2D input variable is distributed across a 'RAM array' (multiple RAM devices) which provides an entire dimension of concurrency. Items in each variable row (or column, depending on the data orientation) can be accessed concurrently across each RAM device—substantially increasing available memory bandwidth with very little implementation overhead. This inherent flexibility of memory customisation with FPGAs underpins our ability to develop novel memory subsystems that are specifically targeted for the kernel under acceleration.

4.3.2.1 *Memory controller design*

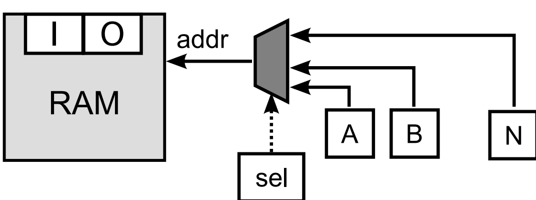


Figure 18: Address control for multiple accesses of a single RAM. A, B, ..N are counters for their respective access location. sel is a counter that iterates between accesses.

Each RAM storage element (whether singular, or in an array configuration) is paired with an associated local controller that moderates the addressing of its port for each and every access. My current implementation embeds counters for each read and write access that is

1 Theoretically, additional data ports can be implemented for greater concurrency, using duplication techniques. Such an approach is useful in certain situations, but is both hard to generalise and limits scalability due to the resource consumption.

managed within the controller, shown in Figure 18. Uniform dependences require only simple counting functionality and is easy to implement. These access counters are multiplexed to the RAM address port, and thus serialised memory accesses can be handled easily by using an auxiliary counter as the select mechanism.

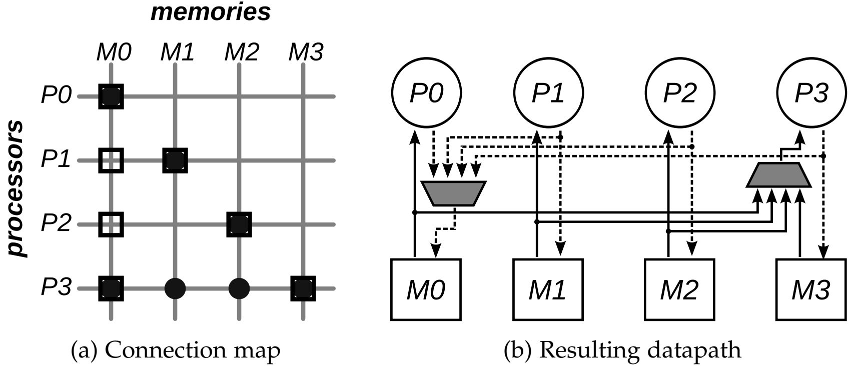
More intricate schemes can eliminate some (or all) accesses, by implementing forwarding of data between PEs, and more closely implementing a dataflow execution pattern. I do not explore the dynamics of these research directions in this work. In the presented examples, my approach has no significant performance impact versus direct forwarding since addressing operations (directly corresponding to accesses) can be easily prepared and still maintain desirable concurrency.

4.3.3 *Register storage*

A register is a clock-controlled singular storage element, implemented natively using FPGA logic resources. Each register supports simultaneous read and write of the element. Once again, we can create a group ('register array') of multiple such registers in a loosely-coupled, logical array. The advantage of using registers over a RAM construct is the greater available concurrency (bandwidth) of potential read/write access to any and all elements simultaneously. However, register capacity on an FPGA device are an order of magnitude fewer than RAM capacity, in terms of realisable data words. Hence, for larger data items we choose them over RAMs only when necessary to achieve higher concurrency.

Registers are well-suited for a data location that is accessed at every logical time step, providing a caching effect to eliminate consistent RAM operations. The same applies to data that is invariant or constant for a period of time, for example, an inner loop dimension. Moreover, data accessed as a function of the processor dimension is realised effectively using an array of registers. This allows the concurrent access to each element in the variable if all processors are active simultaneously, for a given time step, while a RAM would mandate serialised access to the same data.

4.3.4 *Interconnection network*



(a) Connection map (b) Resulting datapath

Figure 19: Model of the internal interconnection network.

The interconnection network between components is the backbone of the architecture, composed of point-to-point links between processing and storage elements. Each link connects a single source (output) to one or more sinks (inputs). Multiplexers are incorporated implicitly where any single sink has multiple sources, with external control signals providing write arbitration. Such a scenario facilitates resource sharing of sinks where time-multiplexed, non-concurrent, data writes are acceptable.

A connection map for each unique kernel statement must be built to capture a complete network.

This network can be modelled and built by building a connection map between all storage and computational elements, conceptually illustrated in Figure 19a. This diagram captures data in two directions: filled circles to represent memory to processor flow, empty squares to represent processor to memory flow, and its possible to have bi-directional flow which is represented by an overlapping square and filled circle. At a high-level this mapping represents a subset of a *cross-bar* like network, which shows that theoretically there should be support for the direct connection of any two points. However, the actual implementation is of a static network, illustrated in Figure 19b. For example, there is a direct connection between P1 and M1 in both directions. Unlike the input of P3 which is multiplexed from the output of all the memories. A control signal must be applied to select which memory should be connected to the input of P3 at any given time instance.

Generalising, the target architecture suggests a PRAM-like model of computation that requires the network architecture to support a, possibly, fully-interconnected communication subsystem that is scal-

able for the size of the device. The network maps well to the physical routing fabric of an FPGA; composed of many local links interspersed with fewer fast global links. Therefore, routing resources are a constrained resource and there is an important balancing act between core logic elements and interconnect utilisation in order to make the most of the device [36]. Estimating how much routing resources should be present in each device is non-trivial and FPGA vendors used statistical methods and Rent's rule [90] to ensure there is 'enough' communication capacity for the amount of implementable logic in a given device. In modern FPGAs this leads to routing resources becoming the dominant area component of the device, which makes it feasible to design the dense and complex networks proposed in this work.

I hypothesise that the breadth or depth of the connection network is not a limiting factor for realising typical accelerators, regardless of the number of processing and storage elements present. This is an idealistic assumption, which generally does not hold for computing hardware, such as networks of GPUs. However, in an FPGA, two consequences could arise: 1) there are not enough routing resources available to connect all elements, or 2) the maximum frequency at which the FPGA could operate at, f_{max}, drops due to propagation delays in the network. Yet, the experiments show that for the targeted applications (nested loops) (1) is rarely the case. In terms of achievable frequency (2), there is a decrease as the FPGA utilisation approaches its capacity, but the drop is to such an extent that this idealisation seems very reasonable. Chapter 7, later in the thesis will demonstrate and elaborate on these findings.

4.4 SYSTEM INTEGRATION

A hardware accelerator implements a specific kernel as part of a larger computing system–whether the system is a collection of discrete devices or even a single system-on-chip. As such, accelerator-host connectivity (as per Figure 14) for both data and control communication interfaces is abstracted. Separate to the accelerator itself, additional hardware is required to implement the external connections, signalling, and any intermediary buffering. Buffering in the data interface can be tuned for streamed data and burst transmissions. Fi-

nally, a generic FIFO interface can then be presented to the accelerator itself.

The data interface is modelled according to the following characteristics: peak aggregate bandwidth (b_{peak}) available between the accelerator-host, the connection contention ratio (α) that describes link availability for accelerator-host transfers, and inherent overhead ratio of communications (ϵ). The former can be expressed as an absolute number determined by physical characteristics of the underlying network, while the last two are values in range 1.0 (zero contention or overhead) down to 0.0. The contention and overhead ratios can be determined experimentally and amortised over time. The product of these parameters is an amortised sustained bandwidth ($b_{sust} = \alpha \cdot \epsilon \cdot b_{peak}$) which represents the realisable data throughput of the connection.

Latency of data between the two end-points is always prevalent. At this point the model largely ignores this latency, as it does not have a large impact for the accelerator paradigm of my focus. In the case of streaming applications the latency is omnipresent, causing a persistent processing delay, but it does not affect functionality. For off-loaded data processing the latency is a negligible component of the overall communication time for large data packets.

Putting this into perspective, a typical system may use the PCI Express (PCIe) interface standard. A common PCIe v2 8-lane implementation provides a peak throughput of 4 GiB/s in a single direction. It is shown in [5] that real world throughput (b_{sust}) of over 3.5 GiB/s could be realised. In the experimental evaluation, a safe design assumption is made of two double-precision words at 200 MHz per direction (3.05 GiB/s).

Naturally, the available bandwidth will play a role in the overall performance of the system. In addition, many interconnects support dedicated bi-directional channels for concurrent communication in either direction. Further, to fully saturate the available bandwidth, for communication-bound systems, communication should be overlapped with computation. For simplicity, both these communication optimisations are not considered in this work, but are natural optimisation opportunities, which can make the approach even stronger.

4.5 CONTROL MECHANISMS

The accelerator control unit is a, simple, finite state machine (FSM) that implements the desired computation loop. It has been generalised for any accelerator datapath and works to provide the read, write, and execute phase for each computational schedule. This is complemented with any application-specific pre- or post- loop tasks.

At a high-level this control architecture affords the benefits of in-field, or even run-time configuration, and simplifies the synthesised control hardware with a minimal impact on frequency. Alternatively, I have experimented with embedding the execution schedule directly as states within the control FSM. This removes the need to load an execution schedule but potentially creates a very large and complex state machine (e.g., thousands of states) that is difficult to optimise and synthesise.

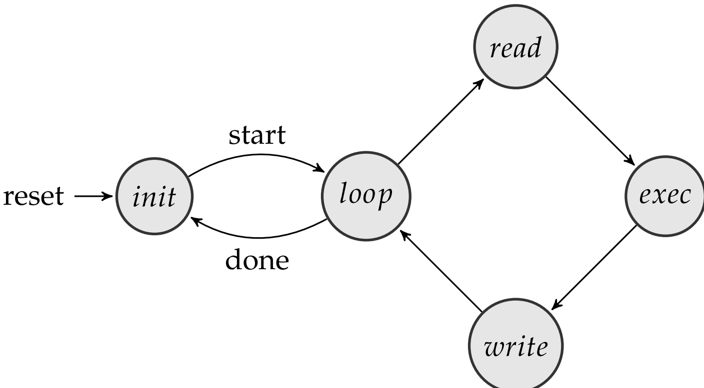


Figure 20: Simplified accelerator FSM state diagram.

Overall, the state diagram looks like Figure 20. The *init* state represents a reset state which advances into the loop meta-state. In actuality this is a hook for the pre- and post- loop initialisation tasks and, importantly, to check the execution progress. These tasks could potentially be contained within separate states or merged (and possibly, duplicated) to optimise performance and minimise cycle-delays. The crux of accelerator execution is performed within the read-execute-write (REW) cycle:

READ PHASE Any steps required to access necessary data from the involved storage elements is performed. It is important that each required data element is guaranteed to be stable and ready for capture. For a RAM, this means to correctly assert the address location

so that valid data contained at that location will be output by the next clock cycle.

EXECUTE PHASE All processing elements involved in the computation at this time instance are activated by the use of an enable control signal.

WRITE PHASE Data writes are actually performed implicitly, as the output valid signal of a PE cascades into the write enable control signal of the configured corresponding storage element–automatically triggering the write operation. Therefore, this state merely provides a delay to synchronise execution.

Control schedule entries

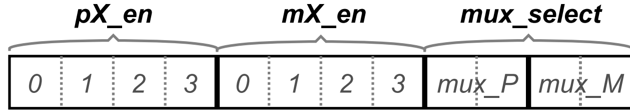


Figure 21: Example control entry structure for Figure 19b.

The execution and control schedule is stored within an embedded RAM. This is partitioned into PE, SE, and multiplexer segments. Each schedule entry encodes the operational state of all the relevant datapath components for each logical time step following strict execution of the REW cycle.

Figure 21 shows the structure of a control entry for the datapath of Figure 19b. There are eight single-bit *enable* fields for each of the four processors (pX_en where X is in {0..3}) and four memories (similarly, mX_en). These fields act as flags which indicate whether the associated entity is to be activated in that given logical time step. Each field is directly connected to the corresponding datapath component and latched according to the FSM. For instance, the memory flags are used to increment the respective RAM address counters, while the processor flags are asserted during the execution phase to capture processor output to memory. Two 2-bit fields (mux_p and mux_m) encode the four possible positions for each entity-class multiplexer; these selection signals are applied at the beginning of each logical execution step. It is feasible to expand this basic structure for more fine-grained control of datapath elements if necessary, for example, to facilitate greater control over updates to memory addressing.

The amount of information stored in the control memory directly corresponds to 1) the length in time steps of the execution and 2) the number of datapath components. At the implementation-level, I do not consider how and when the control RAM is loaded.

4.6 PIPELINED RESOURCE SHARING

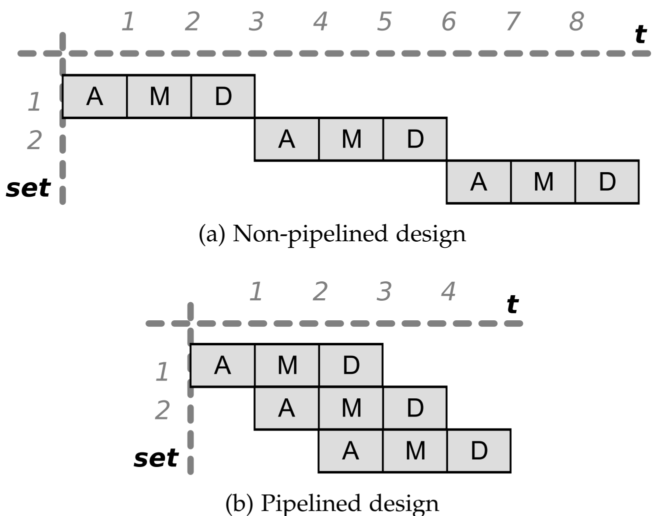
Pipelining is a key technique to improve the efficiency of a hardware design. This is achieved by substantially increasing throughput for any given execution unit by sharing it in a time-multiplexed manner. Conversely, the number of execution units required to achieve a desired throughput can be reduced. It requires that computational resources (the PEs) are specifically designed to support different stages of computation, whereby the input of each stage is the output of the preceding stage. Multiple sets of input data can be *in-flight*, within the PE, simultaneously such that each data set is contained within a different stage of the pipeline. Valid output is only produced once a data set has traversed each and every stage. The output of proceeding sets of data will follow in each further stage of execution. Since each stage is relatively short, the potentially sustainable throughput is improved.

Given the properties of pipelining, it useful both for resource sharing or performance objectives. Generally, we can say it improves efficiency of a design.

4.6.1 *Pipeline model*

First, consider the effects of pipelining by defining a formal model for the technique. Consider a function: $f(a,b,c,d) = \frac{(a+b)\cdot c}{d}$. We want to compute this operation repeatedly for a series of data input tuples. A normal, non-pipelined, execution core would contain three floating-point units for each of the three operations: an adder, a multiplier, and a divider. A computation begins with the addition which is performed using the adder unit, this is proceeded with the output passed into the multiplier unit where the multiplication is performed, and similarly the division is performed using the divider unit. All subsequent computations are performed serially in the same way, visualised in Figure 22a; a new computation does not begin until the current one has finished.

Observe that during each internal stage of the computation (addition, multiplication, and division) only one floating-point unit is



(a) Non-pipelined design

(b) Pipelined design

Figure 22: Impact of pipelining on latency and throughput of a computational core. A, M, and D represent the additional, multiplication, and division sub-operations.

active. Pipelining builds on this observation by breaking these three internal sub-operations into distinct stages. Then new input data can be accepted for computation as soon as the first stage is finished, eliminating the idle time that was present in the non-pipelined design, visualised in Figure 22b.

Comparing the two designs it can be seen that the latency for a computation remains the same at 3 time steps. However, throughput is improved tremendously because after the first data set, the consecutive results are produced after each consecutive time step. Assuming there are n data sets to calculate then the total execution time is $n \cdot 3$ versus $3 + (n-1)$ time steps.

In general, when a problem is broken into k stages we let $T_{pipe} = \max(T_1, T_2, \ldots T_k)$, the largest stage duration.All stages should maintain an equal stage duration to ensure consistent operation and dataflow between stages. Thus T_{pipe}, corresponds to the longest stage of the pipeline, dictating the pipeline period. All shorter stages can integrate buffer registers to hold values until the next pipeline cycle.

The initial latency of a PE then becomes: $T_{pipe} \cdot k$, and the throughput for n sets of input data: $\frac{n}{T_{pipe} \cdot k + T_{pipe} \cdot (n-1)}$, which tends towards $\frac{1}{T_{pipe}}$ per time-step when n is large.

It must be stressed that a common side-effect of pipelining is an increase in the overall latency of a computation due to the additional

overhead required within each sub-operation stage. However, this is often minor compared to the overall throughput afforded by the technique. The reader can refer to [31] for further details.

4.6.2 *Pipelined PE design*

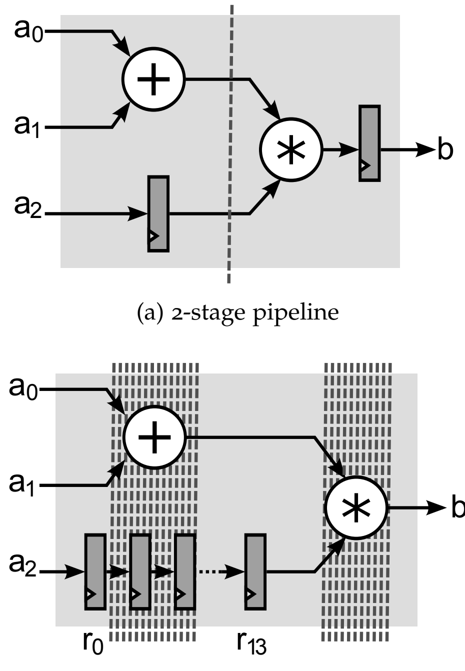
Most PEs can benefit from some degree of pipelining at a low resource cost, leveraging the many embedded register bits on offer within modern FPGA families. In Figure 23 we examine some typical pipelined PE units, for the same mathematical expression: $b = (a_0 + a_1) \cdot a_2$. It is assumed that the latency of an adder is 14 cycles and the latency of a multiplier is 11 cycles. A non-pipelined implementation would therefore have a 25 cycle latency, with a resulting throughput of 1/25 of the clock frequency. The 2-stage implementation is low cost; it requires buffering on input a_2 to maintain its value during execution of the first stage. Buffering is also used on the output-side of the multiplier in the second stage to hold its value as it is shorter than the first, adder, stage. This design has a worse latency of 28 cycles, now, compared to 25 cycles of the non-pipelined design. However, throughput is now 1/14 potentially improved by nearly 1.8x, as the pipeline period is only 14 cycles.

External mathematical blocks are typically fully-pipelined, or can be configured in this way.

The fully-pipelined design assumes the use of fully-pipelined mathematical blocks internally. These are augmented with registers to buffer data for every stage–which is now every clock-cycle of execution–thus register usage is markedly higher. The path for the a_2 input is now registered for 14 cycles in parallel with the addition unit. Overall design latency becomes 25 cycles (equivalent to the non-pipelined design) but now the pipeline period is only 1 cycle—a potential 25x throughput improvement!

4.6.3 *Control entry interpretation*

The proposed interpretation of pipelining is for sharing a PE with multiple sets of independent input data, using time-multiplexing. Modifications to the control strategy and control entries are minor, since this resource time sharing can be implemented independently of the execution schedule and thus the control program. Multiple processor fields in the control entry can map to a single PE unit at different pipeline stages.



(a) 2-stage pipeline

(b) Full (25-stage) pipeline

Figure 23: Two pipelined PE variants based on Figure 17.

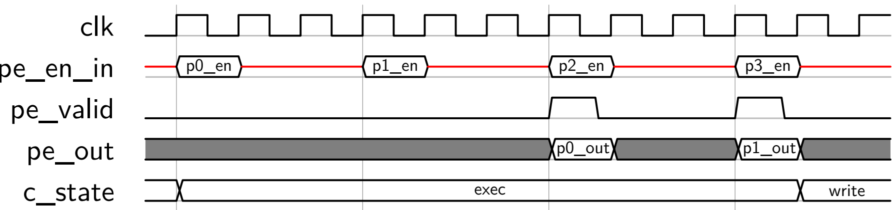


Figure 24: Timing diagram of 4:1 pipelined PE operation.

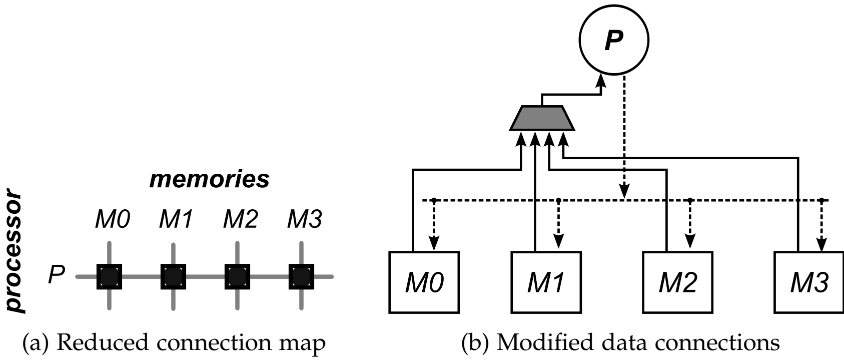
For example, the control entry of Figure 21 is paired with a fictitious datapath consisting of only a single PE (instead of the 4 PEs featured in the original Figure 19b). Pipelining is used to share the resource for the four operations, by improving the throughput of the single physical PE.

Operationally this is depicted in the timing diagram of Figure 24. Each pipeline stage is shown with a vertical grey line, so it can be seen there is a 3-cycle pipeline period and a 2-stage pipeline depth. The depth can be seen by the number of stages between when input is asserted and the corresponding output is produced. The control unit itself requires only minor conditional modifications to be aware of the pipeline period, to know whether to wait or present the next set of data to the PE. Moreover, the *write* state must wait for the entire pipeline to be flushed before proceeding with the next phase of the

control schedule. This is statically derived from the pipeline period and depth.

4.6.4 *Communication arbitration*

To undertake resource sharing there must be the ability to redirect communications from the storage elements to the physical resource. Connections are physically multiplexed to achieve this with the selection controlled by the corresponding data set within the current execution entry. To build this new interconnection network, the original connection map (for example, Figure 19a) is merged by projecting the original processors' connections onto the physical processor (P).



(a) Reduced connection map (b) Modified data connections

Figure 25: Updated interconnection network for shared PE.

The resulting connection map is shown in Figure 25a, with Figure 25b depicting the corresponding datapath. In the latter, only data connections are shown. However, the conditional logic required to implement the original connection table is instead embedded into the SE control signals. These are controlled using multiplexing based on the input data set (`pipeline_counter`), illustrated in Figure 26.

```
-- M0
with pipeline_counter select
  m0_wr_en <= p_valid when 0,
              p_valid when 1,
              p_valid when 2,
              p_valid when 3,
              '0' when others;
-- M1
with pipeline_counter select
  m1_wr_en <= p_valid when 0,
              '0' when others;

-- M2, M3, etc
```

Figure 26: Example of memory write enable multiplexing for M0 and M1. This assumes pipeline execution is in the order of P0, P1, P2, and P3; representing `pipeline_counter` values 0 to 3.

5 POLYHEDRAL KERNEL MAPPING

A goal of this work is to leverage the polyhedral model as a tool to map input codes into a formulation ready for hardware generation as an accelerator. The polyhedral model is an advanced compilation technique that is applied to extract parallelism while scheduling loop nests–akin to semantic analysis versus syntactic analysis of traditional compiler techniques [43]. Computational statements or loop bodies are abstracted as *statements* within this model. The technique focuses only on the control aspect of the code, the scheduling of the aforementioned statement instances.

Polyhedral methods stem from seminal work conducted in [58, 62]. The approach was popularised by its usefulness for the automatic generation of systolic processor arrays that met a set of optimality criteria (for example, the lowest execution time) [61, 81, 82]. Given its rigorous mathematical foundations it enabled the detailed analysis of produced schedules and the ability to predict the characteristics of the produced arrays. Other compiler advances in the form of dependence analysis and loop extraction have enabled the application to general loop nests expressed in traditional procedural languages. Since then the model has gained increasing popularity to address the needs of more general forms of parallelism including multi-core general-purpose processors (GPPs), massively multi-core GPUs, and massively reconfigurable multi-core FPGAs.

Application of this technique can be distilled into three steps:

1. formally model the input codes in a polyhedral representation

2. apply a transformation to this model to produce a new polyhedral model of the data, in the desired context

3. generate target code based on the transformed model

Traditionally, when dealing with source-to-source translation for CPU targets, the final step would be syntactic code generation ready for compilation. However, in this work the generated code is actually used as in intermediary data-structure that captures the target execution schedule that is used to dictate the hardware generation process.

5.1 POLYHEDRAL MODELLING

Given a set of, d, perfectly nested loops, each atomic iteration of a statement can be represented as a point within a polyhedron in $\mathbb{Z}^d$. Each loop defines the extent of the polyhedron in that dimension and the bounds describe the faces. The polyhedron is formed by the set of inequalities defined by the loop dimensions, formally:

$$\mathcal{D} = \left\{ \vec{x} \mid \vec{x} \in \mathbb{Z}^d,\ A\vec{x} \geq \vec{c} \right\}$$

A is a constant iteration matrix, $\vec{x}$ is the iteration vector of loop counters, and $\vec{c}$ is a constant offset or parameter vector. Each statement can be modelled separately as a unique polyhedron.

The input polyhedron is extracted from the static control part (SCoP) of the input code, and is known as the source polyhedron. SCoP is defined in [14] as the maximal set of consecutive statements exclusive of loops that cannot be normalised to an integer stride, and where loop bounds and conditionals depend on constants or invariants within this set of statements; this reinforces the notion that polyhedral analysis focuses on the control of loop kernels. The output polyhedron is known as the target polyhedron. Any dependences between iterations, considered with respect to the sequential execution order, can be overlaid onto the source polyhedron to yield a data dependence graph.

The target polyhedron features dimensions of time and space thus describing a complete execution schedule. Mathematical functions are applied to map points from the source to the target. But, each mapping must adhere to the established dependences in order to consider the transformation as legal. Automated generation of legal transformations piques considerable research interest, including advances in the broader permissibility of input codes [66].

5.1.1 *Data access vector equations*

Each statement can contain multiple accesses to data (or memory locations, considering a traditional computer): at least a single write and zero or more read accesses. These can be modelled as a vector equation of the form: $\mathsf{Sa_i} = B\vec{x} + \vec{b}$. Where Sa is the statement and i identifies a specific access within Sa. B is a constant access matrix, which embeds the iteration vector of loop counters in $\vec{x}$, and $\vec{b}$ is a constant offset vector. Later in this thesis we consider matrix B as an array of row vectors: $B_\mathrm{I} \equiv \left(\boldsymbol{\beta}^1_\mathbf{I}, \boldsymbol{\beta}^2_\mathbf{I}\right)^\intercal$.

5.1.2 *Data dependences*

Dependence analysis is a well studied topic of compiler research and readers are referred to [3, 12] for more comprehensive coverage. A data dependence occurs between two statements that depend on each other, defined as Bernstein's Conditions:

$$(I(\mathsf{Si}) \cap O(\mathsf{Sj})) \cup (O(\mathsf{Si}) \cap I(\mathsf{Sj})) \cup (O(\mathsf{Si}) \cap O(\mathsf{Sj})) \neq 0$$

In this equation, $I(\mathsf{S})$ is the set of memory locations read by S, $O(\mathsf{S})$ is the set of memory locations written by S, and there is a feasible execution path between the two statements Si and Sj.

data dependence within loops Loop dependence analysis studies dependences within one or more nested loops. This is necessary when attempting to perform parallelising optimisations or modifications of the execution of an input kernel. Polyhedral techniques, naturally, fall within this classification. A *dependence distance* refers to the distance between the dependent iterations. In a single loop this becomes simply the difference of the loop variables between the dependent statements. For multiple loops a vector representation is necessary.

Uniform dependences occur when there is a constant dependence distance between statements. This thesis is restricted to uniform dependences only, as it greatly simplifies the analysis. While this limits applicability of this work, I believe it does not inhibit its relevancy to many codes. Moreover, this restriction is not inherent to the proposed approach, but only to its treatment in this work.

A simple stencil kernel is shown in Algorithm 5.1. It represents a computationally-heavy fragment of an input application that we want to accelerate. This example is used as a vehicle to apply and explain the application of polyhedral modelling in the remainder of this chapter.

Algorithm 5.1 Example nested loop kernel performing a point computation. Where $\mathcal{F}$ is any function, e.g. $\mathcal{F}(x, y) = 0.5 \cdot (x + y)$.

kernel StencilComputation(a)
  for $i \leftarrow 0$ **to** 3 **do**
    for $j \leftarrow 0$ **to** $i+2$ **do**
      $a[i,j] \leftarrow \mathcal{F}(\ a[i,j-1],\ a[i-1,j]\ )$ ▷ S1
end kernel

The kernel is analysed to ascertain the source polyhedron from the loop bounds. The input program features perfectly nested loops with a single computation statement, S1. Each point in the source polyhedron represents an instance of S1 at coordinates given by the value of the loop indices at that step. For this example we compute the following polyhedron for the single statement (S1):

$$\mathcal{D}_{S1} = \left\{ (i,j) | (i,j) \in \mathbb{Z}^2, 0 \le i \le 3 \ \wedge \ 0 \le j \le i+2 \right\}$$

This can also be represented in matrix form:

$$\mathcal{D}_{S1} = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \ge \begin{pmatrix} 0 \\ -3 \\ 0 \\ -2 \end{pmatrix}$$

ACCESS VECTOR EQUATIONS In this kernel, there are three data accesses associated with the statement. They are expressed as access vector equations, I to III, as functions of the loop induction variables. For example, for the first access–the write to $a[i,j]$:

$$\text{S1}_{\text{I}}(i,j) = B_{\text{I}} \begin{pmatrix} i \\ j \end{pmatrix} + \vec{b}_{\text{I}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix}$$

The remaining two accesses II and III, the reads to $a[i,j-1]$ and $a[i-1,j]$ respectively, differ only by constant offset vector $\vec{b}$, namely $\vec{b}_{\text{II}} = (0,-1)^T$ and $\vec{b}_{\text{III}} = (-1,0)^T$.

DATA DEPENDENCES It is assumed that the dependences for this input program have been computed upstream and therefore can be easily annotated onto the source polyhedron. In this example there are two flow (read-after-write) dependences: (1) between the write of $a[i, j]$ and then the read of the same value at the next step of j ($a[i, j] \rightarrow a[i, j-1]$) and (2) between the write of $a[i, j]$ and then the read of the same value at the next step of i ($a[i, j] \rightarrow a[i-1, j]$).

All dependences in this example are uniform, thus the kernel exhibits a static data access pattern.

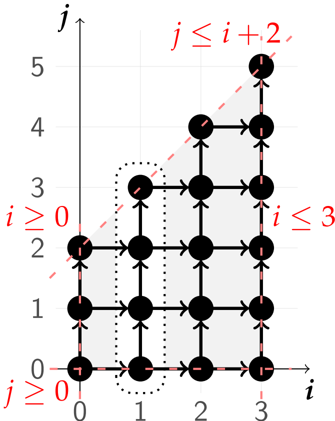


Figure 27: Source polyhedron (as a function of loop indices).

Figure 27 depicts the complete source polyhedron for Algorithm 5.1, with the data dependences overlaid. For the source polyhedron the axes represent the two loop dimensions, labelled with the corresponding induction variables i and j. The dashed lines enclose the polyhedron and are labelled with the corresponding loop constraints.

5.2 SCATTERING FUNCTIONS

Scattering functions are used to transform all points (the statement instances) from the iteration space co-ordinate system into the target polyhedron, on a time-space co-ordinate system that is amenable to hardware-oriented code generation. The objective of the transformation is based on specific design criteria; generally, the desirable outcome is to maximise parallelism while minimising execution time, resource consumption, or unfavourable memory access patterns (that

may incur penalties due to poor cache performance or memory hierarchy awareness).

A transformation is defined as a set of affine mappings enclosed as a scattering function: $\theta(\vec{x}) = T\vec{x} + \vec{t}$. For each instance, $\vec{x}$, within the source polyhedron an affine transformation is applied, with T a constant transformation matrix and $\overrightarrow{t}$ a constant transformation offset vector [13]. Each row of T and t defines a dimension of the target polyhedron, an axis of the target time-space co-ordinate system.

The scattering function is composed of a time schedule and a processor schedule in the form: $\theta(\vec{x}) = \begin{pmatrix} \Lambda \\ \Sigma \end{pmatrix} \vec{x} + \begin{pmatrix} \lambda \\ \sigma \end{pmatrix}$ where Λ and Σ are the time and processor schedules, respectively. Multiple time and processor dimensions can be specified by expanding the number of rows in the respective schedule. When choosing the scattering function it is quite possible to merge, split, and re-order temporal and spatial dimensions and thus a natural point during compilation to take decisions involving the execution order and parallelisation of a kernel.

LEGALITY OF SCATTERING FUNCTIONS A scattering function is deemed to be legal if and only if the target polyhedron honours all dependences present in the source polyhedron. A legal mapping can be visualised when all the transformed dependence vectors are pointing *forwards* in all time dimensions. In other words, there are no statement instances scheduled prior to when the correct data has been computed.

This assumes that a statement instance is computed instantaneously at the given time instance.

5.2.1 *Time schedule*

The time schedule defines *when* a statement instance is to be executed. 'Time' is actually a logical, relative, ordering of the statement instances and not an absolute value. While absolute time is uni-dimensional, additional dimensions can be facilitated using a logical interpretation based on the significance of each–that is, a lexicographic ordering [26].

The simplest time schedule is a sequential ordering of the iteration vectors, and therefore the execution order corresponds to that of a sequential implementation. But, this is more restrictive than necessary and statement instances can be reordered if all dependences are

respected. Multiple time dimensions add further flexibility for statement scheduling, useful in more complex applications.

5.2.2 *Processor allocation*

The processor allocation defines *where* a statement instance is executed. Since I target an FPGA then there is no defined topology that must be followed; we are free to add complexity to the architecture with new dimensions. In practice, a single linear array of 'processors' is an adequate and useful real-world implementation policy.

A processor is interpreted differently to a traditional GPP; a processor (or more accurately, 'processing element' in our terminology) is defined as a hardware execution unit capable of an atomic computation of a loop body statement.

Making processor allocation a separate dimension creates flexibility around resource allocation. For instance, when dealing with hardware, the ability to limit resource consumption is a key attribute. This is directly proportional to the number of processors that are instantiated. Moreover, multiple dimensions can provide processor resource modulation by employing a 'strip-mined' allocation policy [13] with little change to the overarching methodology.

5.2.3 *Transformation of a stencil kernel*

Scattering functions can be applied to transform the example point stencil kernel, Algorithm 5.1, into a hardware-ready execution schedule. The scattering function maps all statement instances from the iteration space co-ordinate system into a temporal and spatial one. Two different transformations are presented that result in two unique mappings, shown in Figures 29 and 30. The axes of the target polyhedra are now p and t, the logically available processors and time, respectively. The new, transformed, dependences are overlaid and flow in the same direction as the time axis proving the legitimacy of the transformations. Four statement instances are enclosed in a dotted field to highlight the effect of different scattering function mappings from the source polyhedron of Figure 27.

wavefront parallelisation transformation For uniform loop dependences, common with stencil kernels, a *wavefront* traversal

of the iteration space can be used to expose the parallelism. The wavefront of execution is in the direction that is between planes of the dependences defined by using the dependence vectors as the normal. Points that can be executed in parallel are on the set of resultant wavefront hyperplanes. This process is illustrated in Figure 28 that shows a subset of the input polyhedron.

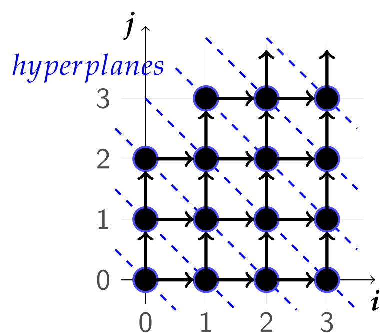


Figure 28: Example of wavefront parallelisation applied to the stencil kernel

To achieve this parallelisation effect, the scattering function employs a time schedule of $\Lambda_1 = \begin{bmatrix} 1 & 1 \end{bmatrix}$. The processor schedule is chosen as $\Sigma_1 = \begin{bmatrix} 1 & 0 \end{bmatrix}$ which dictates that the cardinality of the i dimension ($|\vec{i}|$) determines the number of processors required, as the compute resource, where each instance (i, j) is mapped to processor number i. An offset vector, unused in this case—$\vec{t} = (0, 0)^\intercal$, could potentially be applied to re-position the target polyhedron as desired. The final scattering function is therefore:

$$\theta_1(i,j) = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} i+j \\ i \end{pmatrix} \mapsto \begin{pmatrix} t \\ p \end{pmatrix} \quad (1)$$

Time is expressed as a function of both iteration variables, aligning with the hyperplane: $a \cdot i + b \cdot j = t$ for each statement instance (a, b). An alternative processor mapping could have been chosen, for instance to map statements to processors according to the j variable. However, this case would result in a larger processor dimension size. Figure 29 shows the final target polyhedron after transformation using Equation 1.

NAÏVE SEQUENTIAL TRANSFORMATION The second mapping of the kernel is a naïve, literal, interpretation of the input code. It presents

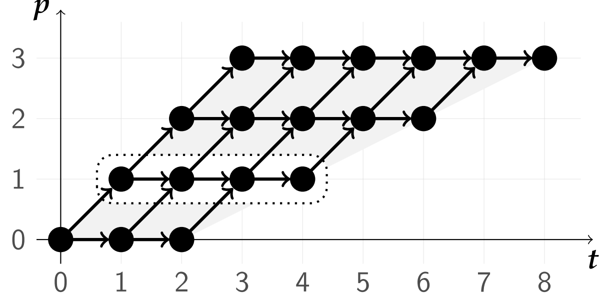


Figure 29: Potential wavefront transformation of the stencil kernel.

a sequential execution of the kernel onto a single processor, depicted in Figure 30. Please note that, in this visualisation, the time axis has been compacted; omitted time instances are treated as a no-operation.

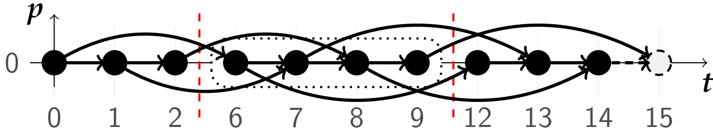


Figure 30: Alternative sequential transformation (execution has been truncated).

This transformation is, perhaps, not representative of a typical accelerator, but a valid and extreme alternative that could be useful in a resource constrained environment. The complete scattering function is:

$$\theta_2(i,j) = \begin{bmatrix} 6 & 1 \\ 0 & 0 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 6 \cdot i + j \\ 0 \end{pmatrix} \mapsto \begin{pmatrix} t \\ p \end{pmatrix} \quad (2)$$

Here a unique time instance must be assigned for every statement instance. This is achieved by segregating time into bins of i steps. We choose a stride size of 6 because that is the greatest number of statements that could be present within an iteration of the i-loop, thus ensuring there is no overlap. The j counter provides inherent ordering within the stride. All instances are forced to use a single processor of constant index 0.

5.3 MULTIPLE TARGET DIMENSIONS

Scattering functions do not have to be mapped to just two dimensions. Ultimately, a real-world hardware execution will only have two phys-

ical dimensions (t and p), however, additional dimensions simplifies scattering function design and enhances the ability to handle more complex scheduling policies.

Multiple dimensions (whether in time or space) provide new planes within which to schedule statement instances. This work assumes that any additional dimensions extend either physical time or space. For *real* execution, there can be only a 'single' physical time dimension and thus all dimensions must be somehow serialised. While it is possible to have many physical spatial dimensions, this is also serialised to a single dimension to simplify the treatment in this thesis.

5.3.1 *Lexicographic time ordering*

Serialisation of multiple time dimensions is achieved by composing the logical time dimensions into a time-stamp vector, and then mapping it to absolute time based on a lexicographic ordering of all time-stamps. Lexicographic ordering of a vector simply means that each dimension has precedence over the the next when considered from left-to-right. An analogy to this is a time-stamp vector of hours and minutes. All minutes within any hour must elapse before any minutes in the next hour. For two dimensions this can be expressed as:

$$(a,b) \prec (a',b') \text{ iff } a < a' \vee (a = a' \wedge b \leq b')$$

Therefore, it results in an ordered iteration of all time dimensions much like a typical sequential iteration of a nested loop. Applied to the stencil kernel (Algorithm 5.1) the following scattering function creates a sequential transformation:

$$\theta_3(i,j) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} i \\ j \\ 0 \end{pmatrix} \mapsto \begin{pmatrix} t_1 \\ t_0 \\ p \end{pmatrix} \tag{3}$$

Compared to Equation 2, this scattering function maps the i loop to t_1, outer, time dimension and the j loop to the t_0, inner, time dimension. Therefore, all statement instances are assigned a time-stamp vector equivalent to the iteration vector: (i,j).

DEEPER LOOP NESTS Thus far the examples have featured only a two-level nested loop. Deeper nesting is another use-case for multiple

time dimensions. This creates different scheduling opportunities for parallelisation and processor reuse. Certain loops can easily be isolated to a different execution plane to maintain execution legality but ignoring that loop variable in localised scheduling decisions.

Algorithm 5.2 Example 3D nested stencil kernel.

kernel CUBESTENCILCOMPUTATION(a)
  for $k \leftarrow 0$ **to** 4 **do**
    for $i \leftarrow 0$ **to** 4 **do**
      for $j \leftarrow 0$ **to** 4 **do**
        $a[i,j,k] \leftarrow \mathcal{F}(\ a[i-1,j,k],\ a[i,j-1,k]\ a[i,j,k-1])$
end kernel

For a three-dimensional stencil kernel, shown in Algorithm 5.2, applying a hyperplane to expose all available parallelism may not be desired due to the large number of parallel statement instances. Although the k outer loop carries a dependence it can be ignored by moving into an outer time-dimension. Instead, a localised parallelisation can be applied on only two dimensions and then scheduled within an inner time-dimension. This is captured in the scattering function:

$$\theta_3(i,j) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} k \\ i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} k \\ i+j \\ i \end{pmatrix} \mapsto \begin{pmatrix} t_1 \\ t_0 \\ p \end{pmatrix} \tag{4}$$

Based on the lexicographic ordering, each instance of $(t_{1,}t_0)$ where $t_1 = k$, must be executed before any instance of $t_1 = k+1$ and likewise for the inner t_0 dimension. Therefore, the outer time-dimension adequately carries and preserves the dependences of the k-loop. Instead, the k-loop repetitively executes the inner parallel schedule. In a practical context this technique would result in the generation of a smaller accelerator core that is reused iteratively by overarching control software. This is a first glimpse into the hardware/software partitioning trade-offs that can be explored.

5.3.2 *Scheduling multiple loop bodies*

The use of polyhedra and auxiliary time dimensions are equally useful to schedule kernels that have multiple statements within different distinct nested loop bodies. Matrix factorisation kernels are a class of algorithms that exhibit this characteristic, an example of which is Algorithm 5.3:

Algorithm 5.3 Matrix factorisation kernel featuring multiple statements (in separate iteration domains)

```
kernel MatrixFactorisation(a)
    for k ← 0 to 3 do
        for j ← k + 1 to 4 do
            a[k, j] ← −a[k, j] / a[k, k]                ▷ S1
        for i ← k + 1 to 4 do
            for j ← k + 1 to 4 do
                a[i, j] ← a[i, j] + a[i, k] ∗ a[k, j]   ▷ S2
    end
end kernel
```

Each statement is modelled as a separate polyhedron, and they are considered and analysed separately with different scattering functions. Polyhedral code generation involves a scanning operation that merges the target polyhedrons. For this example, the following domain equations are obtained for S1 and S2:

$$\mathcal{D}_{S1} = \{(k, j) | (k, j) \in \mathbb{Z}^2, 0 \le k \le 3 \wedge k + 1 \le j \le 4\}$$
$$\mathcal{D}_{S2} = \{(k, i, j) | (k, i, j) \in \mathbb{Z}^3, 0 \le k \le 3 \wedge k + 1 \le i \le 4 \wedge k + 1 \le j \le 4\}$$

Two distinct scattering functions need to be developed for each domain; however, a common target co-ordinate space is employed. Once again, we increase the dimensionality of the target polyhedron to simplify the derivation of appropriate scattering functions. Applied to this kernel the following pair of scattering functions for each statement could be chosen:

$$\theta_{S1}(k, j) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{pmatrix} k \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} k \\ j \\ 0 \end{pmatrix} \mapsto \begin{pmatrix} t_1 \\ t_0 \\ p \end{pmatrix} \quad (5)$$

$$\theta_{S2}(k,i,j) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} k \\ i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} k \\ j+1 \\ i \end{pmatrix} \mapsto \begin{pmatrix} t_1 \\ t_0 \\ p \end{pmatrix} \tag{6}$$

Two time dimensions are utilised once more. Again, the outer k-loop is mapped and isolated as the outer time-dimension that iteratively loops through the inner execution. Statement instances from S1 and S2 are both scheduled within the t_0 dimension, however, there is a dependence that must be honoured. A constant offset of 1 is applied to θ_{S2} to delay statements by one time step, so that the S1 statement is executed first. This logical execution of the target polyhedron is depicted in Figure 31, including the context-switch to transition between the dimensions.

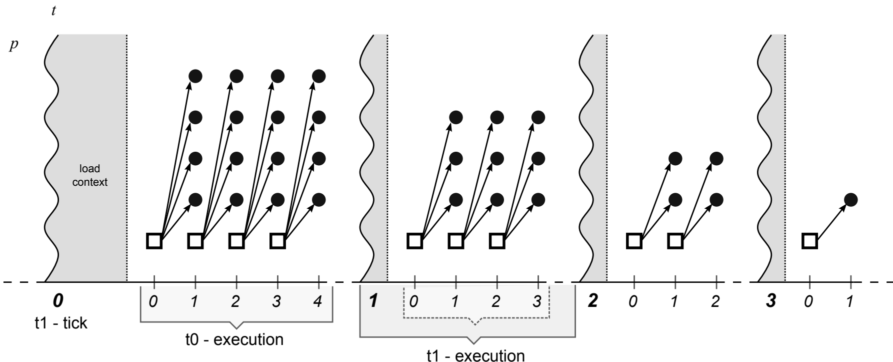


Figure 31: Execution schedule for the matrix factorisation accelerator. *Logical* time flows left to right. t_1 requires a context update before t_0 can execute sequentially. A square represents an S1 instance, while a filled circle represents an S2 instance.

5.3.3 *Virtual processor dimensions*

So far the benefit of increasing the dimensionality of 'time' have been shown. The same considerations can be applied to extend the processor dimensionality to create auxiliary logical processor dimensions. These dimensions can be used to profound effect as a mechanism to re-use computational resources (the processors), optimally in-conjunction with data pipelining.

All example transformations presented have assumed that processors are unconstrained, arbitrarily mapping statements to the processor di-

mension. Obviously this is not a realistic assumption for a hardware-based target since there is a limit to the amount of resources available for processing elements.

DIRECT SCATTERING FUNCTION CONSTRAINTS An alternative approach to implementing resource constraints is to directly embed the constraint within the scattering function. A carefully chosen scattering function can place all statement instances within the desired target constraint. For example, one valid option for the stencil kernel is:

$$\theta(i,j) = \begin{bmatrix} 1 & 2 \\ 1/2 & 0 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} i + 2 \cdot j \\ i/2 \end{pmatrix} \mapsto \begin{pmatrix} t \\ p \end{pmatrix} \tag{7}$$

Figure 32: Direct scattering function modification.

Figure 32 shows the target execution schedule for which only two processors (processing elements) are required. Choosing an appropriate and effective scattering function potentially becomes a non-trivial task as the complexity of the input polyhedron(s) increase. Still, for certain classes of problems, like stencils, it could be sufficiently automated.

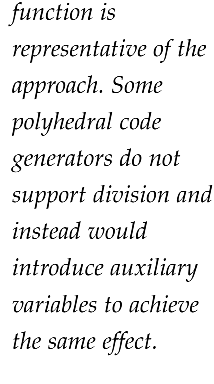
INCREASING PROCESSOR DIMENSIONALITY Another approach to capture resource constraints is to expand the scattering function to incorporate additional virtual processor dimensions. New rows are used to create an ordering vector of execution within a resource. As with increasing time dimensionality, a execution ordering vector can be derived. Actually, the additional processor dimension is used to time-multiplex usage of the physical processor resource. This logical ordering ensures that statements are executed in a correct order that honours any dependences.

Given, t, the time dimension and then p_{phy} and p_{virt}, representing physical and virtual processor dimensions, respectively. Then order-

ing of instances would follow the ordering vector (t,p_{phy},p_{virt}), where the p_{virt} dimension represents the precedence of execution specifically for the associated p_{phy} resource. This precedence is evaluated in normal ascending numeric order.

Creating the extra dimension as part of the scattering function is relatively straightforward. The original mapping to the p dimension can be retained as 'ideal' logical virtual processor dimension, p_{virt}. An integer division can be applied to this to create discrete bins for the available physical resources, p_{phy}. It follows that the divisor can be modified to vary the resource constraint as appropriate. An example application of this to constrain the original stencil scattering function (Equation 1) to use only two processors is:

$$\theta(i,j) = \begin{bmatrix} 1 & 1 \\ 1/2 & 0 \\ 1 & 0 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i+j \\ i/2 \\ i \end{pmatrix} \mapsto \begin{pmatrix} t \\ p_{phy} \\ p_{virt} \end{pmatrix} \tag{8}$$


The scattering function is representative of the approach. Some polyhedral code generators do not support division and instead would introduce auxiliary variables to achieve the same effect.

To physically facilitate this new schedule we extrapolate the concept of the logical tick to abstract the multiple time-steps needed for each virtual processor. That is, a logical tick has a duration of $\lceil |p_{virt}| \ / \ |p_{phy}| \rceil$ physical computational time-steps, to accommodate the case of maximum virtual processor usage within this kernel. Figure 33 shows a visualisation of this schedule. Dashed lines at an offset to the discrete time intervals illustrate the time-multiplexed resource sharing of the two physical processors.

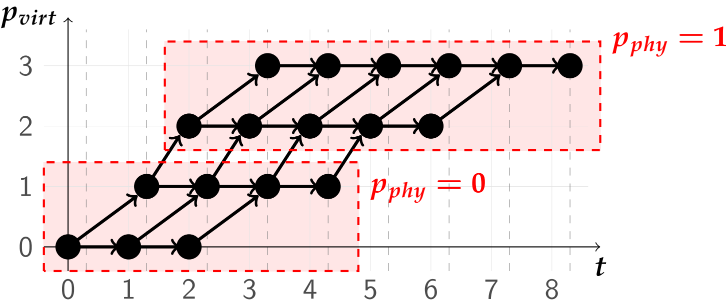


Figure 33: Resource constrained stencil schedule, showing two physical processor (p_{phy}) regions and execution offset.

Unlike directly deriving a constrained scattering function, this approach is universally applicable with little effort. Automation can be achieved very easily and the constraint is easily customisable. As seen in Section 4.6, the datapath modifications necessary to support this

method of resource sharing involves simple multiplexing. Moreover, because we are assured there are no present dependences, we can immediately leverage resource pipelining to improve throughput. Pipelining would change the performance dynamics such that it would be advantageous, due to the low cost, to increase the number of virtual processors mapped to a physical processor.

5.4 TILING THE TARGET POLYHEDRON

Eventually, no matter how large the size of the FPGA device, a limit of feasibility will be reached because of the inability to scale the accelerator design. This could be either directly related to available resources, as a result of diminishing achievable clock frequencies for the design, or–more likely–the combination of both. Techniques such as resource sharing help to alleviate some of these design limitations and improve efficiency but at a cost of increasing network complexity within the datapath. As the internal networks grow, they negatively impact the necessary clocking and limit the achievable frequency. There are some low-level solutions to mitigate these effects, for example, crossbar, multi-stage interconnection hierarchies, and network-on-chip solutions [65], however, in many cases it will still be impossible to avoid reaching the feasibility ceiling of the monolithic schedule.

Ultimately, the next level of scalability can be achieved only by breaking the problem (specifically, the target schedule) into smaller pieces and computing them separately. These smaller chunks are referred to as schedule tiles, which themselves are dependent based on the underlying ordering of the schedule. This adds a level of coarse-grained computation to the overall kernel execution. Within this context an accelerator module can be reused wholesale to execute the different schedule fragments. Moreover, it is possible to duplicate accelerator modules wholesale and apply a distributed computing paradigm, employing concurrency at the module-level.

As an example of tiling the stencil kernel, consider Figure 34. The execution schedule has been derived from the first parallelising scattering function that has been presented (Figure 29, Equation 1), assuming four processors. On top of this, four corresponding tiles form a set that covers the entire schedule.

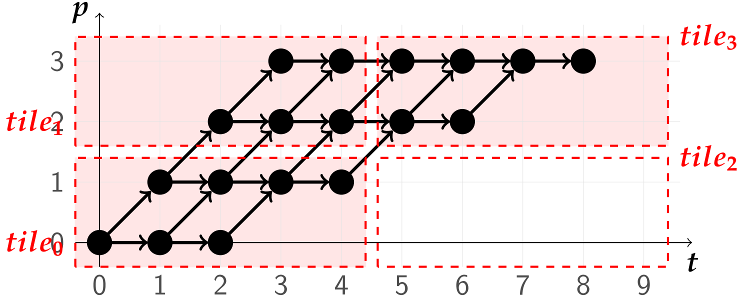


Figure 34: Execution of stencil kernel with parallel scattering function (Equation 1) overlaid with schedule tiles.

5.4.1 *Tile characteristics*

It must be, first, stressed that this tiling approach works on the target polyhedron and not the iteration space like conventional compiler loop tiling. The technique of applying tiling is the same, however.

Assume the target polyhedron can be described as a matrix of schedule dimensions $(\mathbf{T}, \mathbf{P})$ where the number of processors is $|\mathbf{P}|$ and the duration of execution is $|\mathbf{T}|$. This can be partitioned by a tile defined by (d, b) of depth and breadth, respectively, corresponding to $\mathbf{T}$ and $\mathbf{P}$. The number of tiles necessary to fully cover this space becomes: $\left\lceil \frac{|\mathbf{T}|}{b} \right\rceil \cdot \left\lceil \frac{|\mathbf{P}|}{d} \right\rceil$.

A tile can be defined by maximal processor requirements and maximal duration of execution, in logical time-steps. The processor dimensionality must be consistent between all tiles, as all modules are assumed to be equal in hardware configuration. Thus they must include the maximum amount of processor resources (all types and quantities) that could be present in a schedule tile. The same applies to the time dimension, which directly corresponds to the physical quantity of memory storage. The maximum number of computations that can be performed within a tile (the upper-bound on tile execution time) is limited by the data supply—which is determined by the capacity (depth) of the RAM devices.

IMPLICATIONS FOR HARDWARE GENERATION Using a tiling methodology is minimally disruptive to the core concept of applying scattering functions as the mathematical framework for mapping between algorithmic kernels and hardware designs. It breaks the direct rela-

tionship between the two, and in particular the target co-ordinate system is no longer representative of a final accelerator module.

However, in practise, this distinction can be made independently of a user–and, ideally, could be automated by intelligent tools. A set of accompanying tile constraints can be applied to formulate the appropriate amount and configuration of tiling, separate to the underlying desired execution schedule and scattering function. The latter schedule becomes primarily used to expose parallelism and the interplay of data between processors and storage.

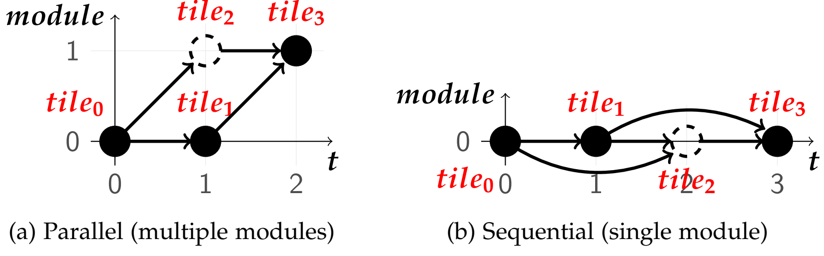
5.4.2 *Execution strategies for tiling*

The process of tiling breaks down the execution schedule into rectangular pieces but with it comes an added level of dependences. The target co-ordinate system is a direct function of time and thus tiles must be executed in the correct sequence to honour this. Simply put, once again an iteration vector can be assigned to each tile based on a new tiling index: (T, P) where T is the index of the tile along the length of the schedule (time dimension) and P is the index of the tile along the breadth of the schedule (processor dimension).

Due to the nature of the tiling, each tile has a direct dependence on the preceding tile in both directions, that is $T_iP_i \rightarrow T_{i-1}P_i$ and $T_iP_i \rightarrow T_iP_{i-1}$. This dependence mirrors the stencil kernel, and it has already been show that a wavefront execution pattern can be applied for parallelism, or a typical sequential execution can be used. Both strategies are depicted in Figure 35. Naturally the former strategy is an unbounded approach with respect to resources, since it assumes that tiling modules could be infinite in number at any given time step. In practice, however, given the independence of each tile, within the confines of a particular time instance, simple scheduling techniques (for example, round robin) can be applied.

Note that $tile_2$ is an *empty tile* as there are no statement instances scheduled within it. For this reason it is not necessary to be executed, and featured as a hollow dashed circle in Figure 35.

Between execution of any tiles there must be a communications context switch. This entails transfer of all current state information of the executing kernel (that is, the data memories). This is the key to correct execution by maintaining data. To practically achieve this, it is assumed that tiling is used in-conjunction with a global controller (as



(a) Parallel (multiple modules)

(b) Sequential (single module)

Figure 35: Execution strategies for schedule tiles.

proposed to handle multiple scattering function target dimensions), which can co-ordinate and distribute the execution of tiles as per the desired strategy.

5.4.3 *A note on tile shape*

It is valuable to think about the shape of the tile and how it can be optimised. In the general case, given that each tile execution entails context switching overhead, it makes sense to maximise the tile size and conversely minimise the number of tiles needed. The critical bounds are two-fold: the size and spatial distribution of embedded RAM, and the availability and distribution of processing elements (or embedded DSP blocks). Moreover, a parallel execution strategy may favour parallelism over per-tile-throughput, given that data transfers can be overlapped with computation. These are currently open-ended optimisation problems that could be solved within a more complete tool-chain.

5.5 Practical considerations

This chapter has explored the polyhedral model and its use to mapping algorithmic kernels into execution schedules. However, some further considerations are required to formalise the use of this approach for hardware generation—which is the primary application and focus of this work.

5.5.1 *Acceptable input*

Figure 36 represents the grammar of a legal input kernel that can be accelerated, given the proposed hardware model. It is assumed

the kernel is an outer-loop which can contain any number of nested loops or assignment statements. The basis of an assignment statement is typically a compound mathematical expression or a data copy between variables, these assignments are more generally referred to as *statements*.

Each variable reference within a statement is either a scalar or an array access that references constants or an affine function of the loop induction variables. Loop bounds are similarly restricted to constants or affine functions of the surrounding induction variables.

Mathematical expressions within a statement are analogous to black-box functions featuring multiple inputs and producing a single output (stored in the assignment variable). Standard integer and floating-point data types are supported. All mathematical constructs are provided by existing hardware cores. Furthermore, it is trivial to include new modules which augment (or improve) the available mathematical functions, allowing for more powerful expressions.

On the surface it may seem that input codes are quite restricted, but it is rich enough to encompass a large range of scientific codes [14] that are compute-intensive and highly attractive candidates for hardware acceleration.

The input language can be any imperative or procedural language, the methodology is independent of this. It is feasible for an accelerator to be integrated into the run-time model of many different programming language environments.

5.5.2 *Typical transformations*

In this work it has been found that simple transformations in-conjunction with higher-level (run-time) control mechanisms are most beneficial. The approach to hardware generation is to focus on one and only one time and processor dimension, thus building a compact and reusable accelerator. This can be controlled and reused in various ways, for example, by iterating through multiple dimensions and applying modular tiling. Therefore it is suggested that scattering functions focus on extracting fine-grained parallelism, where possible, and leverage auxiliary dimensions for simplifying controller implementation.

⟨*kernel*⟩ ::= ⟨*loop_stmt*⟩

⟨*loop_stmt*⟩ ::= `for` ⟨*string*⟩ `=` ⟨*affine_expr*⟩ `to` ⟨*affine_expr*⟩ `do` ⟨*stmt_list*⟩

⟨*stmt_list*⟩ ::= ⟨*stmt*⟩ | ⟨*stmt_list*⟩ ⟨*stmt*⟩

⟨*stmt*⟩ ::= ⟨*assign_stmt*⟩ | ⟨*loop_stmt*⟩

⟨*assign_stmt*⟩ ::= ⟨*variable*⟩ `=` ⟨*math_expr*⟩ | ⟨*variable*⟩ `=` ⟨*variable*⟩

⟨*variable*⟩ ::= variable | array variable reference with indices as an affine function of surrounding loop induction variables

⟨*math_expr*⟩ ::= any ANSI-C style compound expression including the use of arithmetic operators (`+`, `-`, `*`, `/`, `++`, `-`), bitwise operators (`~`, `&`, `|`, `^`, `<<`, `>>`) and common mathematical functions (`sqrt`, `exp`, `inv`, `log`, `atan`, `sin`, `cos`, `abs`, `min`, `max`); where operands can be any variable access or numeric constant

Figure 36: Pseudo-grammar for an acceptable input kernel.

6 ACCELERATOR GENERATION

The preceding chapters explored the proposed FPGA-centric hardware model for the accelerator, and a framework, based on the polyhedral model, for mapping algorithmic kernels to an execution schedule abstraction. Now these pieces can be combined to aid in the process of generating the actual application-specific accelerator, suitable for hardware synthesis.

6.1 GENERATION PROCESS

Overall, the generation process consists of four intertwined phases, shown in Figure 37:

- Generation of processing elements (PE), as black-box modules to implement the algorithmic statements of the kernel
- Generation of storage elements (SE), based on a data layout appropriate to the kernel and fitting with the desired execution schedule
- Generation of controller microcode, that implements a fully-timed state machine for the desired execution schedule; along

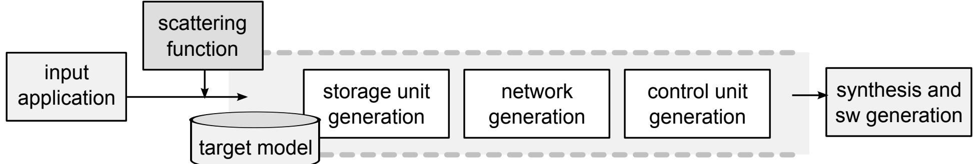


Figure 37: Design flow for accelerator generation.

with any accompanying intermediate control code for run-time processing

- Generation of the interconnection network between PEs, SEs, and the control unit, to enable the correct data movements necessary for the accelerator

Finally, this culminates in a set of synthesisable HDL codes that can be used as part of an external tool-chain to build the complete accelerator module. Run-time support is required to implement additional dimensionality that may be present in the mapping scattering function or when using module-level tiling.

The details of this run-time, or platform, integration has not been considered in this work; it represents an extension to the control generation process. It is a difficult, but feasible, software development task, however, this work concentrates on the hardware generation aspect of the overall process.

6.2 PROCESSING ELEMENT GENERATION

The ability to quickly generate effective PEs is vital to this work. Moreover, it is also useful for related works which may rely on the hardware implementation of specific mathematical expressions as part of a custom design. However, the focus here is on developing PEs that can fit with the hardware model suggested in Chapter 4.

Overall, the end-to-end generation process is as follows:

1. **Expression extraction.** This entails analysing the input code (that is, the specific statement relevant to this PE) and extracting the infix mathematical expression. The specific data accesses are abstracted into generic inputs. In a production system it would be beneficial to pre-process the input code and perform any permissible mathematical optimisations.

2. **RPN conversion.** Convert the extracted infix expression into a list of postfix, or Reverse Polish notation (RPN), tokens. For example, this can be performed with a post-order traversal of the expression abstract syntax tree or with the shunting-yard algorithm.

3. **Building a datapath representation.** Given information about the underlying operators and functions that are supported, a

stack-based postfix evaluation technique can be used to build a circuit description that is essentially an intermediate representation of a valid datapath structure.

4. **HDL generation of complete datapath.** The intermediate representation and operator library are combined with string templating techniques to generate working HDL code. The datapath structure maps conveniently to structural-level HDL code within a generic PE template.

In this section it is assumed that (1) and (2) have been completed, externally. See texts such as [1, 87] for more details on expression parsing, stack processing, and fundamental compiler theory. The focus of this work is on building the hardware representation.

6.2.1 *Datapath formulation*

The datapath intermediate representation is developed by using Algorithm 6.1. This extends common postfix evaluation to build and maintain data structures for the operations and signals present.

A traditional algorithm to evaluate an expression works by 1) pushing operands onto a stack, 2) recognising an operator, 3) removing the most recent operands from the stack, and 4) evaluating the operation, then pushing the result back onto the stack. This is repeated until all postfix tokens have been considered and results in the stack containing a single, final, value: the result of the expression.

In this work, I have modified this algorithm to focus on signals as operands instead of values, to build the datapath circuit. The circuit begins with a set of external input signals (that would themselves hold the values) which are pushed onto the stack (L5–7). When operators are encountered the necessary number of signals are popped (L8–12). These signals represent the set of input signals for the operation, and a new internal output signal is created which carries the result of the operation (L21). The entire operation, including the operator (functional execution core) and the set of inputs and outputs, is boxed and recorded within the datapath state object (L22). The, newly created, output signal is then pushed on the stack (L23) and the algorithm can continue. At the end of the process, the stack will contain a signal which carries the ultimate output result of the circuit.

Algorithm 6.1 Parse a RPN expression to formulate an intermediate representation a PE datapath

Require: operators and input_sigs input tables

Ensure: builds datapath and signals data structures, indexed by *stage*

```
1: kernel BUILD-DATAPATH(rpn_expr)
2:     stack ← []
3:     stage ← 0
4:     for all token ∈ rpn_expr do
5:         if token ∈ input_sigs then
6:             out_sig ← RECORD-INPUT-SIGNAL(token);
7:             PUSH(stack, out_sig)
8:         else if token ∈ operators then
9:             operands ← []
10:            for i ← 0, NUM-OPERANDS(token) do
11:                operands[i] ← POP(stack)
12:            end for
13:            for all sig ∈ operands do
14:                if n ← STAGE-AVAILABLE(sig) ≥ stage then
15:                    stage ← n + 1
16:                end if
17:            end for
18:            for all sig ∈ operands do
19:                MARK-SIGNAL-CONSUMED(sig, stage)
20:            end for
21:            out_sig ← MAKE-INTERNAL-SIGNAL();
22:            RECORD-OPERATION(token, operands, out_sig, stage)
23:            PUSH(stack, out_sig)
24:        end if
25:    end for
26: end kernel
```

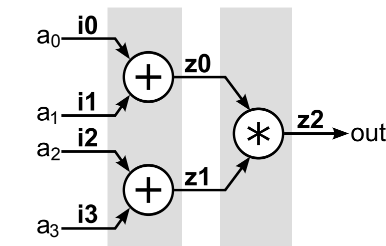
	Token	Stack	Datapath structure	Signal updates
1	A	$i0$	→ [/]	$i0 : 0$
2	B	$i0$ $i1$		$i1 : 0$
3	$+$	$z0$	→ [add | i0, i1 | z0 | /] → [/]	$z0 : 1$
4	C	$z0$ $i2$		$i2 : 0$
5	D	$z0$ $i2$ $i3$		$i3 : 0$
6	$+$	$z0$ $z1$	→ [add | i0, i1 | z0 | •] → [add | i2, i3 | z1 | /] → [/]	$z1 : 1$
7	$*$	$\emptyset$	→ [add | i0, i1 | z0 | •] → [add | i2, i3 | z1 | /] → [mul | z0, z1 | z2 | /] → [/]	$z2 : 2$

Table 7: PE datapath generation processing example.

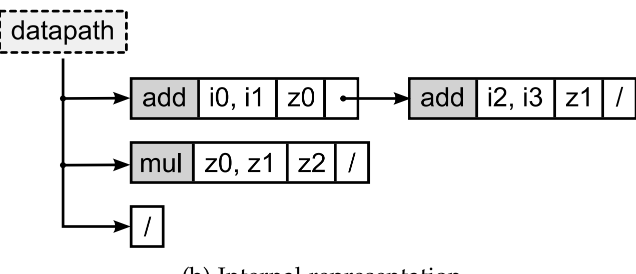
Algorithm 6.1 constructs the datapath as a circuit represented by a linked data structure. To illustrate this process, consider an example expression: $out = (a + b) \cdot (c + d)$. In postfix form the expression becomes: $A\ B\ +\ C\ D\ +\ *$, and this is shown in a worked example of the generation algorithm in Table 7. Figure 38 shows the generated output.

These steps are illustrated in the worked example of Table 7. The 'token' column represents the next token from the input expression to process. The 'stack' column depicts the state of the stack after the step has been completed, with new input and intermediate signals being pushed on and operands being popped off. The 'signal table' column shows additions to the internal signal table, in this case it captures the stage at which the signal is available for use. Finally, the 'datapath structure' column illustrates the growth of the datapath as new operations are added, as a sorted stage list of an operation list.

Timing is critical for any circuit design. Since each operator can take a different amount of cycles, time-based dependences are implicitly introduced. In the general sense these are resolved by using registers to buffer the values of intermediary signals until the set of inputs to a execution core are all valid. In Algorithm 6.1, this is achieved by breaking the datapath into a set of execution stages via an auxiliary stage counter. Each operation which is fed by a signal that would not



(a) Circuit visualisation, showing two stages



(b) Internal representation

Figure 38: Example of generated PE datapath.

be ready is delayed into the next stage (L13–17). Due to the natural ordering of a postfix expression, this is a convenient and reasonable decision. For example, referring to Table 7, the last step adds a multiply operation that depends on signals $z0$ and $z1$ which are not available till stage 1, thus triggering a new stage list to be added to the datapath structure.

An alternative strategy is to use cycle accurate timing, which could be found in a fully-pipelined PE unit. This modification is not shown, but can be achieved by extending the counter technique with single-cycle stages.

The circuit diagram shows the datapath as would be expected, with two adders and one multiplier. Two shaded boxes represent the two computational stages, reflected within the datapath structure. Signals are labelled in bold, with an i-prefix for inputs and a z-prefix for internal signals.

The datapath structure is a linked list of a list of components within each processing stage. Each component element is a tuple of operator function, input signals, and output signal.

6.2.2 *HDL code generation*

A templating engine is used to perform the actual HDL code generation. A template is simply a plain-text file (or string) that contains

```
{% for sig in signals %}
    signal {{sig.name}}: std_logic_vector(S-1 downto 0);
    {% if sig.registered %}
    signal {{sig.name}}_reg_out: std_logic_vector(S-1 downto 0);
    {% endif %}
{% endfor %}
```

Figure 39: Pre-processor template of VHDL signal declarations.

embedded 'tags' to signify blocks of text that must be substituted or processed. The engine loads the template and processes it using accompanying contextual information. For example, given a simple template to say 'hello': `Hello {{name}}`, the tag field, `{{name}}`, is replaced with a person's name during processing.

Jinja2 [84], a Python based engine, is used in this work. It provides an API for invocation and extensibility, allowing for injection of the necessary intermediate data from earlier processing steps along with HDL-specific functions. To complement, Jinja2 includes a powerful domain-specific language for data manipulation, control-flow, and other semantics within the template itself.

Such text processing techniques are very amenable to HDL code generation, particularly using a structural coding style which corresponds closely to the datapath intermediate representation. An accelerator template defines the basic structure of an accelerator, within which are blocks that are substituted with specific implementation details. For example, Figure 39 shows a template fragment that declares all signals within a VHDL `architecture` block. Figure 40 defines input registers as processes that capture new information when an `in_valid` signal is asserted high. Figure 41 instantiates all necessary operator modules and maps their input and output ports with the appropriate circuit signals. This template assumes ports are labelled in alphabetical order (which is the case for Altera floating-point component declarations).

Simple loops and conditional logic dominate the brunt of the template logic needed for the actual VHDL generation. Data processing filters for binary, hexadecimal, and integer conversions and padding are also commonly used.

```
    -- input registers
{% for sig in input_signals %}
    reg_input_{{sig.num}}: process (clock)
    begin
        if (clock'event and clock = '1' and in_valid = '1') then
            {{sig.name}} <= in_{{sig.input_name}};
        else
            {{sig.name}} <= {{sig.name}};
        end if;
    end process;
{% endfor %}
```

Figure 40: Pre-processor template of VHDL register instantiation for inputs.

```
    -- operator instantiation
{% for op in operators %}
    op_inst_{{loop.index}}: {{op.core_name}} port map (
        clock => clock,
    {% for sig in op.inputs %}
        data{{loop.index|alphabet}} => {{sig}},
    {% endfor %}
        result => {{op.output}},
    );
{% endfor %}
```

Figure 41: Pre-processor template of VHDL operator module instantiation and mapping.

6.3 storage element generation

Storage elements store necessary computational data during kernel execution. Remembering that a typical accelerator can be modelled as a derivative of a PRAM machine (refer to Section 4.2), the storage architecture is crucial to enabling high-performance. The bandwidth provided by the storage sub-system is directly proportional to the throughput of the accelerator, assuming unlimited processing resources.

Using an FPGA provides the flexibility to design a truly bespoke memory architecture that is tailored for an algorithmic kernel, rather than traditional approaches that map to rigid buffers or fixed memories. The purpose of the memory subsystem is to provide access to all required data at each and every time step of the target polyhedron. Therefore, where reasonably possible, I favour a design whereby data accesses are performed in a single physical clock cycle, thus maximising potential bandwidth. To this end, the focus is on the *access pattern* of a variable and its relation to the target polyhedron dimensions–temporal and spatial–to ascertain the most appropriate storage element for it.

The prerequisites for the proposed process is: 1) the memory access functions of the input kernel, and 2) the desired scattering function that describes the target execution. In the remaining section it is assumed that these are calculated externally, during polyhedral analysis of the kernel.

6.3.1 *Storage element analysis*

Determining the number and type of SEs necessary to satisfy the algorithm is non-trivial. I propose correlating the memory access vectors for a particular variable of the input kernel with the applied scattering function. This establishes the nature of the access pattern, and hence the type of SE needed to satisfy the variable.

An access vector is a multi-dimensional index that refers to an element within a corresponding multi-dimensional variable. In the (sequential) input program the variable access is made within a loop nest of multiple dimensions, but for physical hardware generation we limit ourselves to a single temporal dimension (even though the scattering function could have many such dimensions). It is assumed

We make this assumption to simplify hardware generation; within the polyhedral model it is trivial to interchange dimensions as desired.

the scattering function already embeds the desired serialisation of time, where only the inner-most (or least significant) time dimension is considered for hardware generation. The inner-most dimension is also the most crucial, for fine-grained parallelisation.

The following criteria is applied for storage analysis:

- **Variable accesses that benefit from concurrency**: multiple accesses to different locations per time step. An array of RAMs would be most appropriate to provide multiple access ports.
- **Variable accesses that are a linear function of the time:** a single access to any location per time step. A single RAM would suffice since there is no bandwidth pressure.
- **Variables accesses that are completely static across a time dimension:** references to fixed location(s) across all time steps within the time dimension. Registers (singular for one location or an array for many locations) provide an efficient storage solution.

To make this classification I seek to answer three questions about each variable access: (1) Is any access dimension a function of a time dimension? This indicates accesses are performed as a function of this dimension. (2) Is any access dimension a function of a processor dimension? This indicates multiple data items are needed per time unit, thus benefiting from concurrency. Finally, (3) is any access independent of any time or processor dimensions? This indicates that these dimensions are not relevant within this schedule.

This evaluation is made by performing a dot product operation to test *alignment* of the original iteration space with the target coordinate system. All vector pairs of the memory access function matrix ($B = (\beta_1, \ldots, \beta_i, \ldots)^T$) and scattering function dimension matrix ($T = \begin{pmatrix} \Lambda \\ \Sigma \end{pmatrix}$) are tested. If the dot product shows orthogonality then the access dimension is independent of the scattering dimension under test. Table 8 summarises the correlation that is sought and serves as the basis for the proceeding analysis.

The function to perform the memory layout in shown in Algorithm 6.2. This is applied to all the polyhedral statement domains of a kernel. All the accesses of a statement (both read and write) are iterated over (L2–7) to build a map data structure of accesses to a storage

$\beta^i \cdot \Lambda$	$\beta^i \cdot \Sigma$	Storage Element
0	0	Register
0	non-zero	Array of registers
non-zero	0	RAM
non-zero	non-zero	Array of RAMs

Table 8: Storage layout taxonomy, based on any access function dimension (β^i, $1 \le i \le N$, with N being number of rows of B) with the scattering function dimensions (Λ and Σ) within T.

Algorithm 6.2 Build a memory layout for an input polyhedron and accompanying scattering function.

Require: scattering function (T) and access function (B) matrices
Ensure: builds layout data structure, indexed by access variable

```
1: kernel MEMORY-LAYOUT(T, accesses, B)
2:     for all a ∈ accesses do
3:         for all β^i ∈ B[a] do
4:             λ^i ← β^i · Λ
5:             σ^i ← β^i · Σ
6:         end for
7:         layout[a] ← SELECT-STORAGE-ELEMENT(max(λ), max(σ))
8:     end for
9: end kernel
```

element. L3–5 perform the aforementioned dot product computation that is used to verify the correlation between the current access vector (β^i) and the scattering function dimension (Λ or Σ). These scattering dimension sub-calculations are reduced using a max function which tests for the existence of a non-zero element, and then are used as the input for Table 8 (L7). To put this in context, a worked example of this process is shown in Table 9 for S1 with scattering function 5 of the matrix factorisation kernel (Algorithm 5.3).

The run-time complexity of this algorithm is $O(n^2)$ and it is repeated for all kernel statements. In practice, the run-time is likely insignificant due to the relatively small dimension sizes for the number of accesses and the depth of loop nesting.

RAM PLACEMENT The motivation for utilising RAMs is to provide a convenient, low-cost, resource. It can be scaled in an array to provide

ACCESS	ACCESS VECTORS	$\cdot\Lambda$	$\cdot\Sigma$	STORAGE ELEMENT
I: $a[k,j]$	$\begin{pmatrix} 1 & 0 \end{pmatrix}$	0	0	
	$\begin{pmatrix} 0 & 1 \end{pmatrix}$	1	0	
	max	1	0	RAM
II: $a[k,k]$	$\begin{pmatrix} 1 & 0 \end{pmatrix}$	0	0	
	$\begin{pmatrix} 1 & 0 \end{pmatrix}$	0	0	
	max	0	0	Register

Table 9: Application of memory layout algorithm for statement S1 in the matrix factorisation kernel. Λ and Σ are the vectors of the scattering function time and processor dimensions, respectively.

a dimension of concurrency. Therefore, a test with the time dimension (Λ) will map the accessed variable to a single RAM if there is no processor correlation (Σ) or an array of RAMs if there is. The latter case results in the distribution of data into multiple RAMs with each connected to a separate PE for that access. Available bandwidth is increased, yet the design complexity is only minimally increased.

Whether the RAM, or RAM array, is configured in a row-major or column-major orientation is important for maintaining concurrency. Technically, row versus column ordering is applicable only to the first two dimensions of a multi-dimensional array and is also dependent on the underlying language's indexing: `variable[column][row]` or vice versa. More generally, the appropriate orientation for the target RAM element is determined by correlating the algorithmic index dimension with the scattering function dimension which is a function of time. The matching dimension is used as the major orientation dimension for the storage element.

REGISTER PLACEMENT On the other hand, registers are useful both for storing time invariant data–eliminating repeated accesses, and to provide greater flexibility for rapidly changing data. Embedded registers provide the greatest flexibility when designing the memory architecture with almost limitless potential bandwidth and random accessibility. However their use is purposefully limited, since its recognised that they are relatively scarce. Instead, registers are used anywhere there is no time dimension correlation, either as a single

element or in an array if there is also a processor dimension correlation.

6.3.2 *Handling multiple dimensions*

In this work it is suggested that, for the purposes of generating an accelerator, the focus should be on a specific time and processor dimension. This ethos is maintained during the selection of storage elements. Only the relevant dimensions are considered when performing the analysis. This leads to a simplified design and allows the relatively simple logic to work more effectively.

However, this technique is not applied purely in isolation. Between each execution time dimension a context switch is required to synchronise data between the on-chip accelerator memory and the master global data memory. This entails that each SE within the accelerator must be loaded with the correct set of data items within the inner scattering function dimensions. But this approach presents a potential limitation to performance as the overhead of communications to achieve the necessary memory consistency can impact overall throughput; this is explored in greater detail in the experimental work (Chapter 7).

6.3.3 *Storage element coherence and reduction*

After a pass of the SE analysis algorithm, it is conceivable to have accesses to the same variable mapped to completely different storage elements. The values must be consistent across each storage element to maintain coherence. This is a common occurrence; for example, many kernels deal with data in matrices and so computational statements have multiple accesses to different elements within the same array variable. It is desirable to consolidate the storage elements of these overlapped data accesses–eliminating the duplication, and reducing resource requirements, while still capturing the greatest amount of storage concurrency necessary.

storage element reduction proposition Any direct duplicates can be merged if and only if the storage element is identical in class and content. Furthermore, for any variable storage element class where there is both array and singular implementations, then

S1		S2		
I: $a[k,j]$	II: $a[k,k]$	I & II: $a[i,j]$	III: $a[i,k]$	IV: $a[k,j]$
RAM	Register	RAM array	Register array	RAM

Table 10: Initial SE layout for a matrix factorisation kernel.

one array should replace all other storage elements if and only if 1) the underlying variable structure is the same and 2) the referenced location of the superseded elements is contained in the array.

For example, consider Table 10 showing the layout and accesses for a matrix factorisation kernel (refer to Algorithm 5.3) with a parallelising scattering function applied. The RAMs can be eliminated, consolidating the accesses to the RAM array which is already suggested. However, the register elements cannot be merged as, in this specific case, the singular location reference is not contained within the array. Referring to the same algorithm, during the execution of the loop $i \leftarrow k+1$ so the register array $a[i,k] \Rightarrow a[k+1,k]$ and clearly $a[k,k]$ is outside of this scope.

CONSISTENCY OF STORAGE While designing the memory system, we must be careful to keep each location consistent so as not to introduce an error into the kernel execution. In the proposed reduction technique, merges only occur for storage elements which contain the same location thus avoiding potential inconsistencies. Moreover, if we did not consolidate variable memories then it must be ensured all writes of a variable are broadcast to every SE that contains a copy of the referenced element. For registers there will be an inherent duplication, but the model states that they are, by design, always consistent within the inner-most time dimension; instead, synchronisation can occur between time dimensions.

6.4 GENERATION OF CONTROLLER CODE

Each accelerator includes a control unit that implements the full desired execution schedule. The schedule itself is represented by control schedule entries within a RAM, where each entry embeds all necessary operations at that particular time instant.

The generation of this series of control entries is the topic of this section. The key process involved is the mapping of the desired control schedule into a control flow graph (CFG) that represents the complete accelerator execution schedule. The CFG is, also, amenable to further analysis as part of the hardware code generation of the interconnect generation. With the CFG in place it becomes straightforward to walk this graph to produce a complete, timed, evaluation of the execution. After that, the control entries can be generated trivially.

6.4.1 *Accelerator CFG*

The original algorithm kernel itself can be represented as a CFG; it is oriented with respect to the iteration space. After applying a scattering function to generate a control schedule we have the basis for a modified CFG that must be adapted to our new space-time coordinate system. The accelerator CFG can be generated automatically by scanning the target polyhedron and injecting the datapath design as the context for mapping. `CLooG` [13], a polyhedral code generator, is used in this work to perform the polyhedral analysis and generation of the schedule CFG. Invocation of these steps has been done manually, though it could be easily automated and integrated.

Under acceleration, the input kernel makes references to variables and constants that are now mapped to physical storage elements and computational statements which have been abstracted into processing elements.

A modified CFG grammar that embeds concepts relevant to hardware implementation is utilised and defined as in Figure 42. This is based on two iterator nodes for scheduling: in sequential time (`for`) and across parallel space (`forall`). The argument to an iterator is the iteration variable, directly corresponding to the scattering function dimension, and a start and stop value that corresponds to an integral range. The `max` and `min` and other mathematical functions or operators may be used within the range specifiers and are valid as long as they evaluate to an integral value.

Iterators can be further refined using relational 'filters'. Computational nodes (statement instances) are then placed within this scheduling framework. A computation is an assignment so the left-child is always a storage element. The other children are the required (con-

$\langle cfg \rangle$ ::= $\langle block \rangle$ | $\langle cfg \rangle$ $\langle block \rangle$

$\langle block \rangle$ ::= $\langle seq_iter \rangle$ $\langle block \rangle$ | $\langle par_iter \rangle$ $\langle block \rangle$ | $\langle filter_expr \rangle$ $\langle block \rangle$ | $\langle compute \rangle$

$\langle seq_iter \rangle$ ::= `for` $\langle variable \rangle$ `:` $\langle range_expr \rangle$

$\langle par_iter \rangle$ ::= `forall` $\langle variable \rangle$ `:` $\langle range_expr \rangle$

$\langle filter_expr \rangle$ ::= relational expression composed of literal constants and the induction variables ($\langle variable \rangle$) of the parent iterators

$\langle range_expr \rangle$ ::= an integer pair of lower and upper bounds with an integral stride

$\langle compute \rangle$ ::= sub-tree defining the statement instance as a (destination, function, source) triple

$\langle variable \rangle$ ::= induction variable of iterators

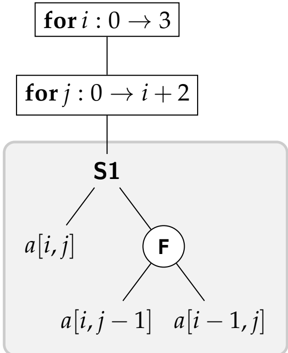
Figure 42: Pseudo-grammar for the modified accelerator schedule CFG.

nected) inputs to the computation. Any node within this scope can be indexed as a function of the outer iterators.

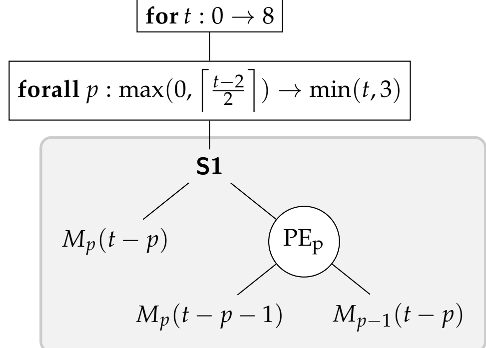
Running the CLooG tool on the stencil kernel with the parallelising scattering function returns the following output:

```
/* Generated from stencil.cloog by CLooG 0.17.0 gmp bits in 0.01s
    . */
for (t=1;t<=8;t++) {
  for (p=max(1,ceild(t-2,2));p<=min(3,t);p++) {
    S1(p,t-p);
  }
}
```

The produced output code corresponds, more or less, to the desired CFG model. CLooG is invoked to set up the scattering function dimensions with corresponding t and p variable names which is mapped to iterator nodes. Likewise, conditional expressions are mapped to filter nodes. This step of the proposed design flow is performed manually, for now, but an abstract syntax tree of the schedule is exposed (`clast`) and can be used to integrate the process within a tool.



(a) Initial: referencing the concrete variable accesses within the input code.



(b) Modified: referencing the selected SEs and PEs aligned to the target co-ordinated system.

Figure 43: CFG for stencil kernel example, based on parallelising scattering function (Equation 1).

6.4.2 *Statement to resource substitutions*

All statements are rewritten with respect to a physical mapping: a PE, and all the data access to all required SEs. Resources are written in the form $R_y(x)$ to represent instance y of resource class R, with an optional location x–useful to describe the offset of a location within a storage element (RAM). The specific instance index and memory location can be an expression based on the scattering schedule dimensions. This is computed easily by using algebraic substitution of the algorithmic loop induction variables with the new time-space dimensions. It is the scattering function which defines the mappings, and this can be directly used to compute the new resources within the accelerator CFG.

Figure 43 shows an example comparison between the initial algorithmic CFG (43a) and the accelerator CFG (43b). The two nested

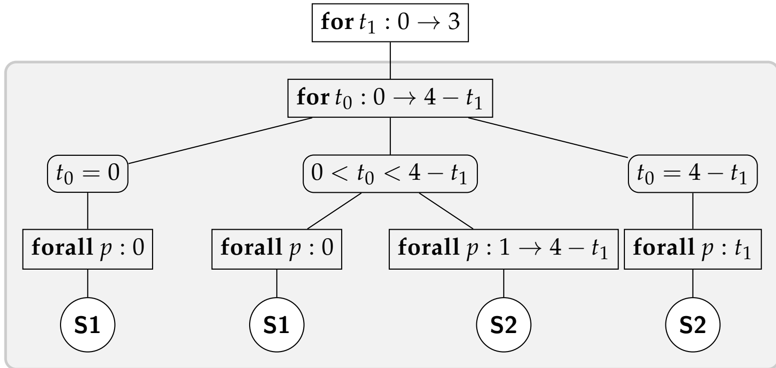


Figure 44: CFG for matrix factorisation example, the shaded sub-graph is executed on the accelerator, while outer nodes are software-controlled.

iteration statements have been replaced with a sequential, outer, time iteration and a parallel, inner, processor iteration. The statement, S1, remains the same in effect, but the data references are rewritten with respect to physical resource and offset as a function of the scattering function dimensions. For example, the access $a[i, j-1]$ is mapped to memory M_p because the first access dimension to i maps directly to scattering function dimensions p. The second access dimension $j - 1$ is substituted with scattering function $j = t - i$, where $i \to p$, which results in the offset $t - p - 1$. This, and iterator boundaries, are extracted directly from the AST generated by the CLooG tool [13].

6.4.3 *Nested time dimensions*

Nested time dimensions are achieved using a co-operative hardware/software approach. The outer dimensions are implemented as software structures within a middleware layer of the host device. This is responsible for controlling the accelerator to correctly implement the underlying inner schedule, which includes transferring of the correct inner schedule (based on the execution state of the outer dimensions) and performing the complete data synchronisation process. Since all control schedules can be pre-computed (and, optionally, stored) this can be modelled as a communications overhead within the enclosing accelerator operation.

Figure 44 is an example of a CFG that is designed in this way. The two consecutive sequential iterators (the `for`-loops) highlight the time dimensions. A normal control schedule is developed for the inner

Algorithm 6.3 Recursive procedure to build the control program (execution schedule) for the accelerator kernel.

```
Require: pre-computed accelerator CFG
Ensure: builds a schedule table defining control program
 1: kernel GENERATE-CONTROL-PROGRAM(CFG)
 2:     schedule ←initialise unmarked t–p matrix
 3:     CONTROL-GEN-DFS(CFG, NIL)
 4:     parse schedule into bit array (binary) format
 5: end kernel

 6: kernel CONTROL-GEN-DFS(v, ctx)
 7:     ctx ← UPDATE-CONTEXT(v)
 8:     if v is a statement then
 9:         for all t ∈ ctx do
10:             for all p ∈ ctx do
11:                 schedule[t][p] ← marked
12:             end for
13:         end for
14:     else
15:         for all child ∈ v do
16:             CONTROL-GEN-DFS(child, ctx)
17:         end for
18:     end if
19: end kernel
```

iterator (t_0) which is executed in hardware, thus enclosed within the shaded box in the Figure. All outer dimensions (only t_1, in this example) are implemented on the host device (typically CPU) within the reconfigurable system.

6.4.4 *Code generation*

With the accelerator CFG in place, the actual code generation of control entries becomes straightforward. Because of the split between hardware and software control logic, there are two paths for code generation: 1) the static binary control entries for the accelerator control RAMs, and 2) the dynamic run-time software control logic. Both paths require a similar traversal of the CFG tree.

For the former, the process is detailed in Algorithm 6.3. It starts by initialising a mutable data structure (specifically, a multi-dimensional array) to represent the execution schedule (L2). The dimensions must be pre-computed according to the algorithm and scattering function. Next (L3), a recursive depth-first search is used to identify all components active within a statement. These are used to build a data structure that embeds a complete timed control schedule. Traversing the CFG tree requires passing through the `for` and `forall` nodes which are used to update the current context of the traversal (L7). This context object captures the scattering function dimension bounds which are evaluated (L9–13) when a statement instance is reached (L8) in order to mark the correct PEs as active within the schedule (L11). Once the schedule has been built it can be parsed and dumped into an appropriate format for binary loading to the control RAM (L4) The binary format has been defined arbitrarily (specified in Section 4.5).

The context object can be implemented as a list of tuples holding bound and strides, for example.

The presented algorithm deals with only the processor schedule, but this same technique is used to generate a memory activation schedule. This differs in the evaluation step (L11) where one must evaluate the correct memory accesses by peeking at the children of the statement node. Once again, the final schedule is saved as a binary format for loading into the accelerator.

It is also possible to perform just-in-time compilation of the control schedules on the host.

The latter part 2, the associated software program, can also be generated using a tree traversal that identifies the outer host-controlled nodes. Software code is generated to dynamically implement the controlling nodes. The body of this code is responsible for loading the appropriate control schedule into the accelerator, as well as any other data synchronisation. In this thesis the control software is the test harness, so the code generation process actually generates behavioural VHDL test stimulus and procedures. For a realistic system, this phase would be realised dynamically, for instance as a run-time driver, or would result in the generation of static application code.

6.4.5 *Virtual processor dimensions*

To handle virtual processor dimensions–that is, pipelined processing elements–is quite straightforward. In fact, minimal changes are required to the CFG and the control schedule generation process. Along with the timed schedule, extra metadata is provided within the code generation context that contains information about the processor con-

```
    -- PE virtual<>physical signal arbitration
{% for i in range(num_virt_pe) %}
    phys_pe_{{i % pipe_depth}}_a <=
        virt_pe_{{i}} when pipe_counter = {{i % pipe_depth}}
    {% if not loop.last %}else{% else %}end if;{% endf %}
{% endfor %}
```

Figure 45: Automatic generation of pipeline multiplexing.

figuration. This includes how many physical processors are available and how deep the pipeline is, that is, how many virtual processors are associated with the physical PE.

From this extra metadata, the code generation process can correctly build the required datapath modifications. This is almost entirely the addition of extra multiplexers to modulate resource usage, as discussed in Section 4.6. Thus, the control schedule itself can remain the same, however, the actual control signals are gated to map the virtual operations to the physical resources. Moreover, the control unit FSM must be aware of these changes and take into account the additional latencies on compute and data write, before proceeding with the next logical time step.

Figure 45 illustrates how this process works. The `pipe_counter` is an internal counter variable, within the datapath, that keeps track of the current pipeline index for the current logical time step. This is used as the selection signal for all pipelining control multiplexers. In this case, modulation is used to trivially map from the virtual processor index set to the physical processor index set in a round robin fashion.

6.4.6 *Module tiling*

Generating a tiled, modular, accelerator implementation requires additional consideration and effort. It builds on the control generation process but must adapt the generated schedule for the desired tiling constraints and different usage model. The overall generation process becomes:

1. Generate the overarching accelerator CFG, as per usual. Use this to generate the complete timed execution schedule without

Algorithm 6.4 Scan the unconstrained execution schedule to build a tiled control program.

Require: unconstrained accelerator *schedule*
Ensure: builds an unconstrained tiled *program* data structure

1: **kernel** GENERATE-TILED-PROGRAM($schedule, breadth, depth$)
2: $program \leftarrow []$
3: **for** $y \leftarrow 0$ **step** $depth$ **to** length of schedule, $|t|$ **do**
4: $instances \leftarrow []$
5: **for** $x \leftarrow 0$ **step** $breadth$ **to** width of schedule, $|p|$ **do**
6: $proc_tile$, $mem_tile \leftarrow$ SLICE-ARRAY($schedule, x, y$)
7: APPEND($instances, (proc_tile, mem_tile)$)
8: **end**
9: APPEND($program, instances$)
10: **end**
11: **end kernel**

any constraints applied. This is used as the basis of the control generation.

2. Generate an additional constrained version of accelerator CFG, that is used only for the purpose of interconnect and hardware generation for the actual accelerator module. Since constraints are applied, the correctly sized accelerator module can be built.

3. The unconstrained execution schedule is sliced into tiles according to the computational constraints (number of processors) and execution depth (capacity of the storage elements). These tiles are used within a software control system to implement the desired tiling strategy.

This approach provides a separation between the generation of hardware (the accelerator datapath) and the control logic (both software run-time control that implements the tiling and the embedded control schedules).

Step 3 of this process is illustrated in Algorithm 6.4. This simple procedure walks over the entire [unconstrained] program schedule, as per the aforementioned control generation process, in strides of the desired tile depth and the tile breadth (L3–5). A list of lists data structure is used to hold the complete tiled control *program* (initialised on L2) and a temporary list holds the tiles for a particular time step or instance of the tiled program (*instances*, initialised on L4). Creating the

t	0	1	2	3	4	5	6	7	8
p_3	0	0	0	1	1	1	1	1	1
p_2	0	0	1	1	1	1	1	0	0
p_1	0	1	1	1	1	0	0	0	0
p_0	1	1	1	0	0	0	0	0	0

Table 11: Partitioned processor control schedule for tiling with $breadth = 2$ and $depth = 5$.

tiled programs is merely a 'slice' of the multi-dimensional schedule which is added to the current instance (L6–7). Here, two tiles are returned *proc_tile* and *mem_tile*, for the processor and memory control programs, respectively.

The final program is an ordered list of unordered lists. Each set of tile instances within the *instances* list is unordered and, so the tiles can be executed in any order within this since there are no dependences between tiles that start at the same control depth. However, the *instances* list itself is ordered and must be executed sequentially. Most commonly, the overall program would be processed either sequentially or in round robin fashion depending on the number of available tiled accelerator units.

Extra conditional checks can be made, at both L7 and L9 of the algorithm, to test for empty tiles or empty instances and exclude them from the program. Empty, in this context, is defined as not performing any computation at any time step (that is, there are no PEs marked to be active).

Table 11 shows an example of the PE execution schedule for the stencil kernel. Assuming tiling has been employed, the light red shaded cells show the shape of the first tile that unbounded schedule has been partitioned into. Note that for some cases, there will be no operations within the tile schedule, for example the bottom-right tile in the table above. It would be safe to skip these steps if and only if there was also no memory activity at that instance.

The actual tile control code maps to a normalised index set with the respect to the scattering function dimensions. That is, the processor schedule is mapped to $\{phy_0, phy_1\}$ and the schedule is timed from 0 till the depth of the tile control memory (in this example 0 to 4)—regardless of the tile's starting indices in the original, unconstrained,

schedule. The reader should be reminded that tiles are defined to overlap the schedule and not the iteration space.

6.5 INTERCONNECT GENERATION

The final aspect of the accelerator generation to be considered is the structural composition of the appropriate interconnection network between all datapath components. This network must be established so there is a valid connection path between each pair of components that must communicate. Overall, this is built by traversing the accelerator CFG (representing the target schedule) and examining the dataflow of all statement sub-graphs. All required connections between each source and sink are added to a network data structure, which can be iterated over to perform HDL code generation.

Statements, within the proposed computational model, lead to two outcomes that require a connection path:

1. **The assignment.** The left-hand side of a statement is a destination storage element which captures the output of a processing element on the right-hand side. This requires a one-to-one connection between the output of the PE to the input of the SE.

2. **The computation.** The right-hand side PE invocation constitutes multiple connections between the PE input arguments (SEs) and the PE computational unit itself. This requires multiple one-to-one connections, for each input argument, between the output of SE and the input port of the PE.

Algorithm 6.5 is used as the basis to walk the CFG and develop the interconnection network. It employs a traditional pre-order depth-first search topology to parse the CFG graph. There are two main tasks that are performed: 1) maintaining the correct contextual information of the iterator nodes, and 2) building connections for all statement nodes based on this context. This context object captures the scattering function dimension bounds with respect to the allowed iteration constructs and is vital when evaluating the statement leaf nodes.

When finding an iterator node, the algorithm collects the associated scattering function dimension and range specification (L2-3). This is merged with the current context and used as the 'new' context for any deeper recursive calls (L5). When encountering a statement node

Algorithm 6.5 Recursively process CFG to build interconnection table for datapath code generation

Require: Complete CFG, where initial node is the root
Ensure: builds network table, indexed by datapath component

```
1:  kernel BUILD-INTERCONNECT(node, ctx)
2:      if node is a Node.ITERATOR then
3:          ctx ← UPDATE-CONTEXT(node, ctx)
4:          for all child ∈ node.children do
5:              BUILD-INTERCONNECT(child, ctx)
6:          end for
7:      else if node is a Node.STATEMENT then
8:          ADD-CONNECTION(ctx, node.left, node.right)
9:          index ← 0
10:         for all input ∈ children (PE inputs) of node.right do
11:             ADD-CONNECTION(ctx, node.right, input, index)
12:             index ← index + 1
13:         end for
14:     end if
15: end kernel

16: kernel ADD-CONNECTION(ctx, dest, src, index = 0)
17:     for all ctx ∈ context do
18:         dest ← EVALUATE(dest, ctx)
19:         src ← EVALUATE(src, ctx)
20:         ADD(network[dest][index], src)
21:     end for
22: end kernel
```

type, real connections are built. Firstly, the output assignment path between SE and PE (L8) and, secondly, a path for each PE input in order (L9–13).

The actual connection creation functionality has been abstracted into a separate procedure (`Add-Connection`), which completely evaluates the context set (as defined by the range constraints) into local iteration instance information of the scattering function dimensions (L17). This local instance context is used to evaluate the actual source and destination component references, since they are typically specified as index functions (L18–19). A data structure holds a map of all components and their inputs. Finally, a reference to the destination object is appended to the source object's input list (L20).

6.5.1 *Stencil kernel example*

This process can be applied to the stencil kernel example (Algorithm 5.1). In this kernel there is a single statement which contains the two dataflow constructs, as discussed:

1. **Assignment**: $M_p(t-p) \leftarrow PE_p$. This statement occurs within a parallel block so all connections are replicated. It describes a one-to-one connection between each RAM and PE, whereby an edge from each node PE_p output to M_p input is added to the connection information for all p. These are shown as red coloured connections in the figure.
2. **Computation**: $PE_p\left(M_p(t-p-1),\, M_{p-1}(t-p)\right)$. It describes a one-to-one connection between the output of M_p and first input of PE_p (shown as blue solid connections), and again between M_{p-1} (the adjacent RAM) and the second input of (shown as blue dashed connections).

Using Algorithm 6.5, we arrive at the basic interconnection diagram of Figure 46.

In a complete accelerator there is additional networking overhead required for all the controllers, memory controllers, and so on. This is not covered in detail, however, it can be automatically generated in a similar fashion.

Furthermore, multiplexers must be implicitly inserted where the connection architecture calls for it. When there are multiple paths

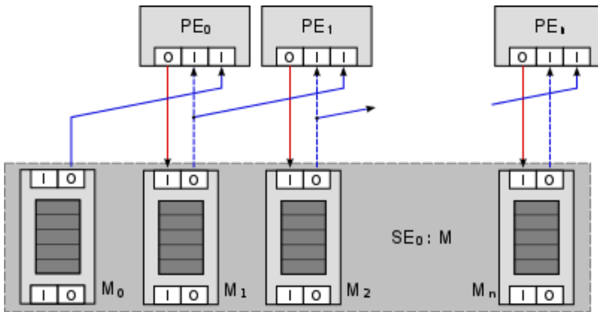


Figure 46: Complete accelerator datapath for the parallelised stencil kernel example.

assigned to a single input port then a multiplexer must be included, along with an associated control signal.

7 EXPERIMENTAL EVALUATION

In the last three chapters I have proposed a systematic approach for accelerator generation in reconfigurable computing systems. This technique utilises the polyhedral model for parallelisation and statement scheduling onto the flexible accelerator hardware model. This chapter serves to demonstrate that the suggested approach is both viable and worthwhile; encompassing an architectural, performance, and subjective analysis of the work.

7.1 INTRODUCTION

To start, an overview of the experimental process will be presented. This paints the high-level picture of what was tested and how it was tested. Performance comparisons are made with an Intel x86 CPU to provide some realistic context into the real-world viability of the suggested approach. Furthermore, an auxiliary accelerator has to work within the bounds of the available interconnection bandwidth (as discussed in Section 4.4), which serves as another viability benchmark.

Primarily, the experimental motivation is to show that the architectural and scheduling concepts work. This is done by examining the 'simple' non-pipelined accelerator implementations for the stencil and matrix factorisation kernels (first introduced in Chapter 5), Section 7.3. Here I can make fundamental assertions about the approach and verify that it behaves as expected, including the method for memory selection and–more generally–the impact of memory architecture on performance. Moreover, design-space exploration and designer usability are key aspects of my work that must be considered.

Fully pipelined accelerator implementations are then presented in Section 7.4 which have a significant positive impact on performance and resource consumption. But, as a side-effect, this also reveals that the design-space for configuration permutations when pipelining is incorporated becomes large and complex.

Following that, I then present experiments for larger problem sizes in Section 7.5. Given that the proposed hardware generation approach results in accelerator size to scale with problem size, this incorporates tiling techniques to break the input data set into more manageable pieces. This is a natural extension to the approach, but does raise some interesting observations with respect for performance per resource, while adding another dimension of parallelism.

7.2 EXPERIMENTAL SETUP

The experiments have been designed as a proof-of-concept for the various methods and underlying architecture that I propose in this thesis. Complete designs are tested using simulation for result acquisition, along with physical hardware verification.

7.2.1 *Inputs*

The accelerators have been experimentally evaluated for the two example kernels that have been presented earlier: the stencil kernel (Algorithm 5.1) and matrix factorisation (Algorithm 5.3). Each accelerator is based on a specific scattering function that dictates its structural and behavioural characteristics.

Four designs are considered across the experiments: **parallel stencil** (scattering function of Equation 1), **single stencil** (scattering function Equation 2), parallel **matrix factorisation** (scattering functions of Equation 5 and 6), and the **parallel stencil** but **with a restricted single RAM** storage element. This final design has been manually developed, and represents a naïve implementation; it adds a point of comparison to highlight the impact of the memory design.

The complete input permutations are presented in Tables 12 and 13. Note that the stencil designs are based on a square input matrix, unlike the example of Algorithm 5.1.

INPUT KERNEL	PARALLELISING TRANSFORMATION	SEQUENTIAL TRANSFORMATION

`for (i=0;i<=3;i++) {`
`  for (j=0;j<=3;j++) {`
`    a[i][j] = S1(a[i][j-1], a[i-1][j]);`
`  }`
`}`
	$\theta_{S1}(i,j) = \begin{pmatrix} i+j \\ i \end{pmatrix}$	$\theta_{S1}(i,j) = \begin{pmatrix} 4 \cdot i+j \\ i \end{pmatrix}$

j, i, 0 1 2 3	p, t, 0 1 2 3 4 5 6	p, t, 0 1 2 3 4 5 6 7

Table 12: Stencil kernel input and accelerator designs. Note: that the sequential transformation target polyhedron has been truncated, and is not complete.

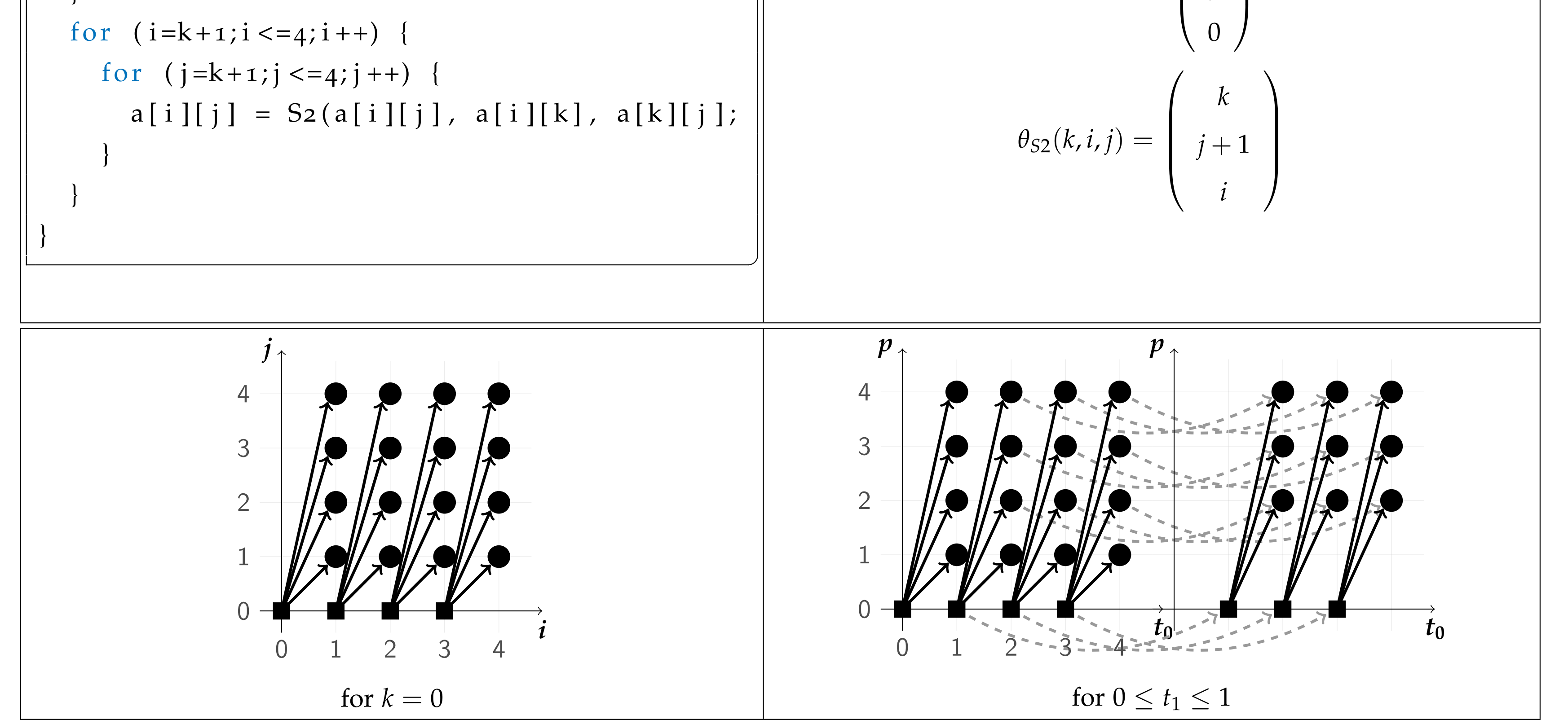


Table 13: Matrix factorisation kernel and accelerator design. Note: the iteration space and output polyhedron are illustrative only; both have reduced dimensionality as labelled below the respective plot. Dashed grey lines on the schedule indicate dependences between t_1 time steps.

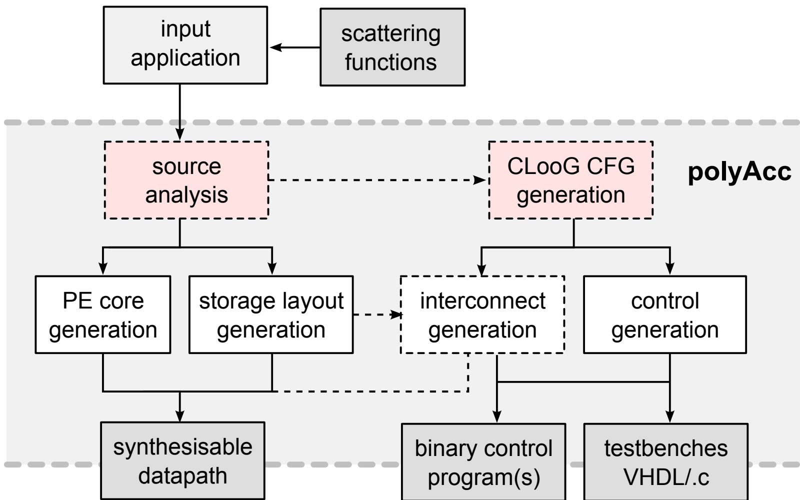


Figure 47: Experimental accelerator design flow using the polyAcc framework. Dashed lines and blocks indicates manually performed steps.

7.2.2 *Process*

The accelerators are semi-automatically derived using an experimental tool-chain, called polyAcc, that implements the methodology and algorithms presented in Chapters 4–6. I developed polyAcc as a vehicle for experimenting and testing the techniques and methods that were formulated during my research, and presented in this thesis. A combination of the tool and some manual design has been used to produce the final implementations.

POLYACC This framework is a collection of proof-of-concept prototypes that implement the methods and techniques presented in this thesis. Figure 47 depicts the accelerator design flow using these tools. Most components have been implemented (based on the algorithms presented in this work) using the high-level Python language, for rapid development and exploration. The lightly coloured red boxes rely on external tools. Dashed components have not yet been implemented and are performed manually.

Firstly, the designer identifies the SCoP in the input application code to be accelerated and selects associated scattering function(s). These serve as the primary inputs. Source analysis is the next phase of

the process. In this work, the input code has been manually scanned and analysed to determine: 1) the input polyhedra, 2) the accesses and access functions, and 3) the dependences. These tasks can be performed using traditional compiler techniques using existing frameworks, such as Polly [46].

CLooG [13] is a polyhedral code generator which performs the brunt of the polyhedral work. It accepts a series of input polyhedrons, scattering functions, and contextual information about the target dimensions and then performs polyhedral scanning to generate an abstract syntax tree of the output polyhedron. The input (`.cloog` problem description) is manually specified and CLooG is manually invoked. The output polyhedron is embedded in a subset of C program code which is manually mapped to Python data structures used in other processes.

PE core generation utilises the input statements to generate hardware cores, automatically for non-pipelined cores. For pipelined cores I have manually modified the generated output. Storage layout generation consists of the selection of the appropriate storage elements for the accelerator. Each accelerator is generated using Python with the Jinja2 templating library [84], based on a base accelerator module which includes the control unit functionality. The datapath comprises PE cores and the storage elements, which are interconnected together. For these experiments I have manually performed the interconnection generation process and embedded it into the datapath generator for each design. The final accelerator consists of a set of VHDL modules that are ready for synthesis.

To complement the datapath, a control program must be generated that is loaded into the control unit memory. This program specifies the complete, cycle-accurate, execution of the acceleration kernel. It is generated using a Python module with inputs of the accelerator CFG which is embedded with the datapath knowledge.

In these experiments, the control program is used to automatically generate tests: 1) a VHDL testbench that embeds the control program along with test stimulus, and 2) a Nios II software application (as a C program) that embeds the control program and test stimulus. The automatically generated Nios II test component includes an extra VHDL hardware module that implements an Avalon bus slave interface, and accompanying device driver, so that the accelerator can

connect to and communicate with the Nios II CPU and application code.

All the accelerator implementations are coded in RTL-style VHDL and incorporating all required external IP cores. Each accelerator supports a fixed 'problem size' which is varied to establish a picture for both small and large problems that demand a large proportion of the underlying FPGA resources. In this evaluation I assume the input is always square matrices and so the problem size maps directly to the dimension size of the input matrix. All data is stored in 64-bit double precision floating point format.

FPGA SYNTHESIS As described above, the polyAcc tool terminates with the output of VHDL code. An external synthesis tool converts this VHDL code to a binary FPGA configuration. In these experiments Altera's Quartus II 10.1 software suite has been used for this task. It is worth noting that the synthesis times for these designs have ranged between 10–200 minutes depending on the complexity of the design (found to be largely dictated by the resource usage).

To some extent, these synthesis times have a bearing on design-space exploration as synthesis must be performed before an accurate picture about attainable frequency and final resource consumption is known. However, the Quartus II tool performs an initial design analysis which took much less time (2–20 minutes) in order to give an initial estimate about resource usage (that is, whether a design will fit and how many resources it needs to be mapped to).

7.2.3 *Target systems*

The VHDL designs have been synthesised to target the Terasic DE4 development board, containing an Altera Stratix IV EP4SGX530 FPGA. The characteristics of this device is listed in Table 14, and they can be used as a reference point for all synthesis results presented in this chapter.

Synthesis has been performed using 'standard' fitter settings with the optimisation level set to 'speed' and a clock frequency target of 200 MHz. Physical correctness verification has been performed using a Nios II SoPC host-system, clocked at 200 MHz, with the accelerator connected to the Avalon bus and controlled and stimulated by the Nios II processor. The Nios II configuration is used as a harness to

FAMILY	Stratix IV
DEVICE	EP4SGX530KH40C2
COMBINATIONAL ALUTS	424960
MEMORY ALUTS	212480
REGISTERS	424960
PINS	888
MEMORY BITS	21233664
M144K BLOCKS	64
M9K BLOCKS	1280
DSP BLOCKS (18-BIT)	1024
PLLS	8
DLLS	4

Table 14: Stratix IV EP4SGX530KH40C2 device information.

run the verification suite and is not representative of a production-ready, stand-alone, accelerator.

Performance testing has been performed using the ModelSim HDL simulator configured with the same 200 MHz clock frequency. CPU benchmarking has been performed on an Intel Q6600 device running at 2.4 GHz. The original sequential kernels (coded in C) have been tested; they are compiled with `gcc-4.4` using `-O2` optimisation running on a Linux 2.6.32 64-bit kernel. Additionally I have tested fine-grained parallel implementations generated with CLooG (using the scattering functions presented earlier) and hand-annotated with OpenMP directives around the parallel loop sections. While performance scaled, all parallel tests failed to match the sequential version in absolute performance.

7.2.4 *Software OpenMP performance*

This thesis has made several comments about the difficulty of writing effective parallel software code. Table 15, which shows performance results for a problem size of 320x320, only reinforces this fact. For this experiment the CLooG generated code, for each kernel, has been manually modified to include `OpenMP parallel for` constructions

to parallelise all loops that are ‘scheduled’ across the processor dimension in the target polyhedron.

The number of threads to be used is specified at run time using the `omp_set_num_threads()` function call.

Unfortunately, using this approach for software does not work as expected. The primary reason is that the transformed code employs fine-grained concurrency. But the OpenMP implementation distributes this work to separate threads which incurs too much overhead to be beneficial. Instead, better performance could be attained by using lower-level SIMD extensions.

7.3 NON-PIPELINED ACCELERATORS

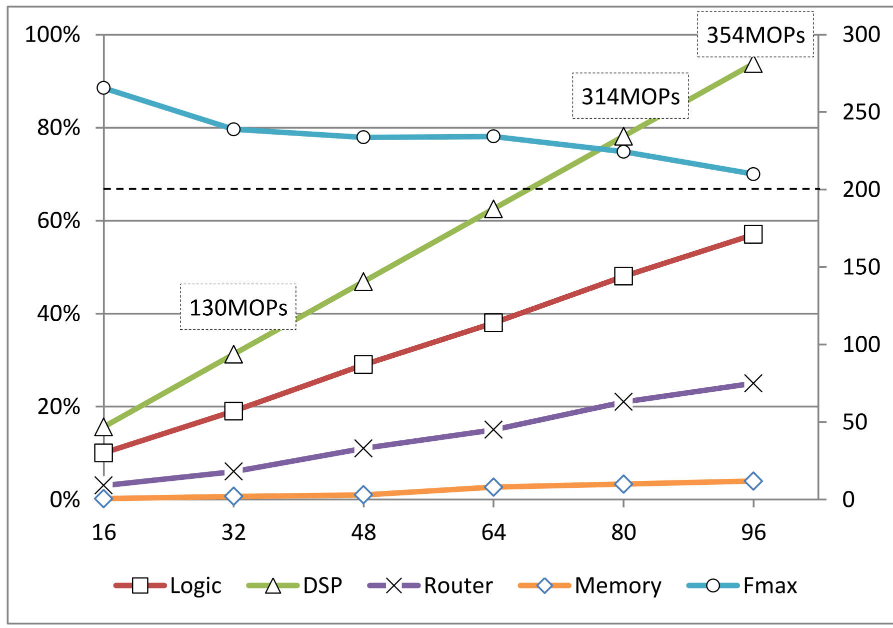
Firstly, the baseline performance and effectiveness of the approach is established by considering non-pipelined control schedules without any consideration for scalability.

7.3.1 *Synthesis results*

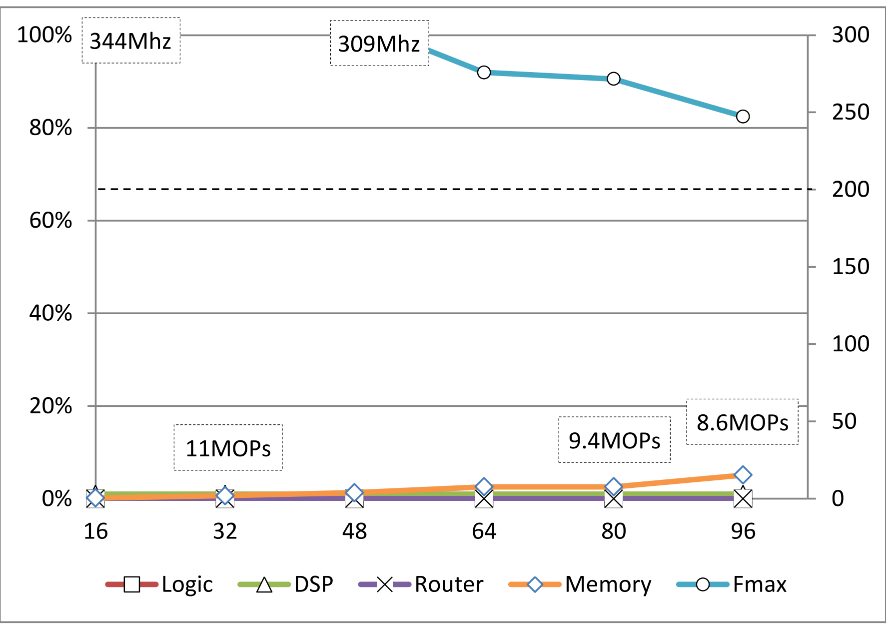
Figures 48 and 49 present the synthesis results for the four accelerators across various problem sizes. All resource consumption units are as a percentage of total device capacity where: ‘logic’ refers to the aggregate device utilisation metric, ‘DSP’ refers to the 18-bit DSP blocks embedded within the device, ‘router’ refers to the reported average router utilisation, and ‘memory’ corresponds to usage of the embedded memory. On the secondary axis, ‘F_{max}’ refers to the predicted maximum clock frequency (MHz) after timing analysis. These values are plotted over the problem size which is varied from 16x16 to 96x96. Remember, the required number of processing elements on the FPGA increases with the problem size except for the sequential stencil design. The latter is fixed to a single processing element .

NUMBER OF PROCESSORS	1	2	4	8	16
Stencil (parallel)	0.80	2.53	3.12	4.29	7.05
Matrix factorisation	16.76	183.45	201.99	272.37	409.89

Table 15: Runtime (ms) of software performance scaling using CLooG generated kernels with OpenMP annotations for problem size 320x320.

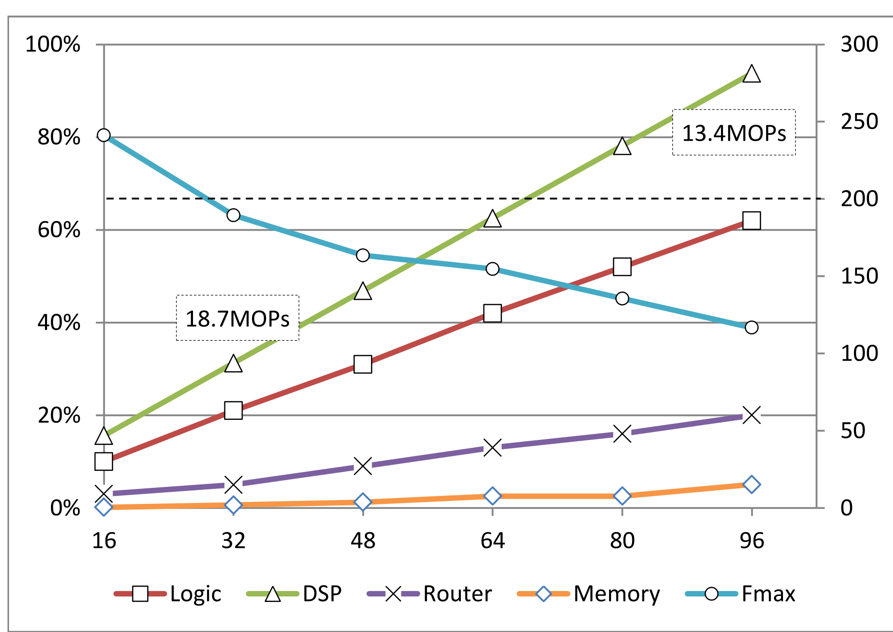


(a) Stencil (parallel)

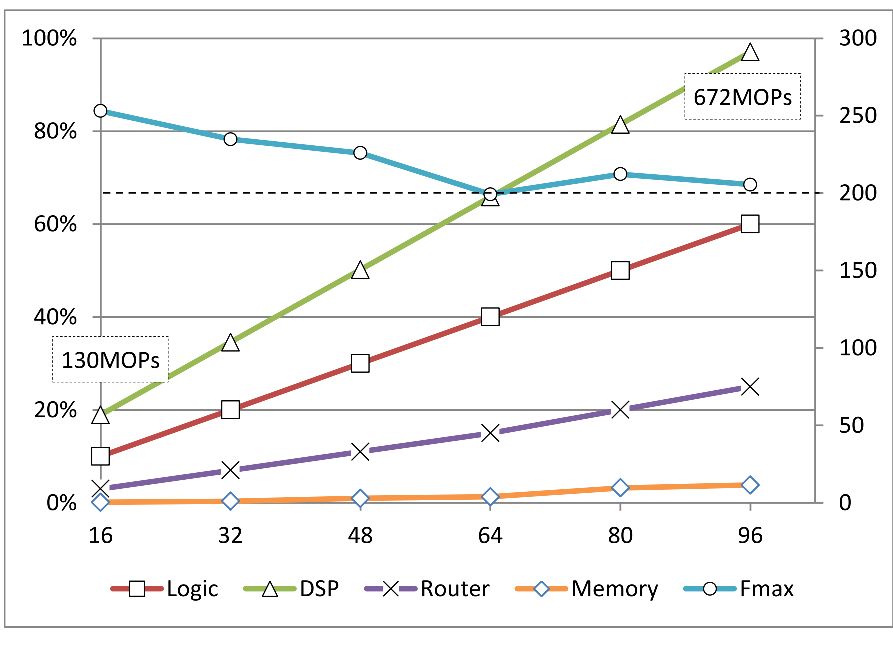


(b) Stencil (sequential)

Figure 48: Synthesis results of HDL accelerator designs. Resource utilisation (%) on primary axis, F_{max} (MHz) on secondary axis over problem size (from 16x16 to 96x96). Selected maximum theoretical throughput inset (millions of operations per second: MOP/s).



(a) Stencil (parallel, single RAM)



(b) Matrix factorisation

Figure 49: Synthesis results of HDL accelerator designs, continued.

Selected throughput results, in millions of operations per second (MOP/s), have been inset. For the stencil kernel this corresponds to the maximum number of stencil computations performed each second on this hardware design (frequency and datapath configuration), when ignoring any communication overhead. A similar metric is used for matrix factorisation except I consider only the inner time dimension of execution and not the complete matrix factorisation kernel. This assumes best-case use of the hardware design to illustrate the impact of frequency and datapath scaling. The throughputs are proportional to the frequency *and* the number of processing elements (that is, the problem size).

7.3.1.1 *Architectural scalability*

Figures 48a and 49b show the synthesis scalability across device utilisation for accelerators based on 'typical' performance-oriented (therefore, parallelising) scattering functions for both input problems. Inevitably, both designs are bound by DSP availability reaching almost 100% utilisation—a common bias for scientific kernels. Moreover, the suggested 200MHz target F_{max} has been met across the board, except for the single memory design, with a roughly linear drop with respect to problem size; router utilisation increases linearly, and is not a concern; other resources, including registers and logic utilisation, show similar characteristics.

From these results it is observed that the parallel designs (Figure 48a and 49b) show good scalability with respect to frequency versus accelerator size; reaching good frequencies with reasonable fall-off for problem sizes within the scope of current-generation FPGAs. In both the parallel stencil design and the matrix factorisation design the maximum frequency drop-off is approximately 20%. Most importantly, it is evident that the biggest problem sizes resulted in the highest throughput despite the slight drop in frequency. So it is meaningful to use the entire device, despite the typical reduction of maximum frequency with the size of the design. Furthermore, it vindicates the communication design, where I intentionally suggest naïve multiplexer-backed connections between components. The presented results prove that it is a tractable approach, having little impact on the studied kernels.

Having that said, it is understood that some computational kernels may necessitate horizontal scaling and use of a coarse-grained many-

accelerator approach in order to achieve higher frequencies. This will be elaborated when tiling is discussed, in Section 7.5.

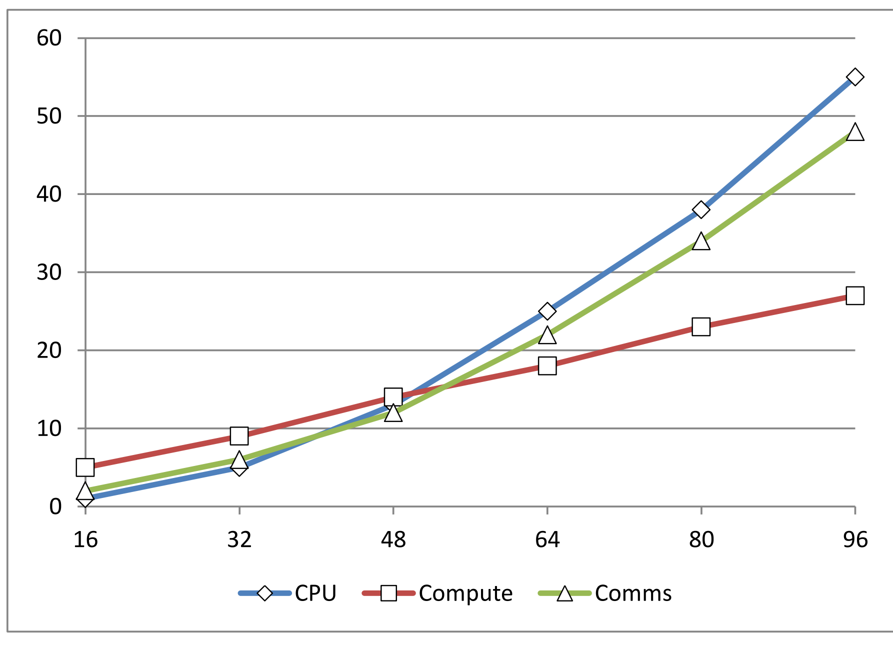
7.3.1.2 *Memory design*

A comparison of the proposed parallel design (Figure 48a) and the single RAM design (Figure 49a) serves to highlight the impact of the storage element analysis on memory design. The latter is meant to represent a naïve accelerator design as it might be implemented manually by a developer that did not consider maximising bandwidth in the architecture. It was created by replacing the, originally generated, array of RAMs with a single RAM (logically partitioned as a 2D array) and then making modifications to linearise the control schedule to account for the change in RAM layout. In other words the designs are conceptually similar but employ different memory configurations (and schedule, as a corollary).

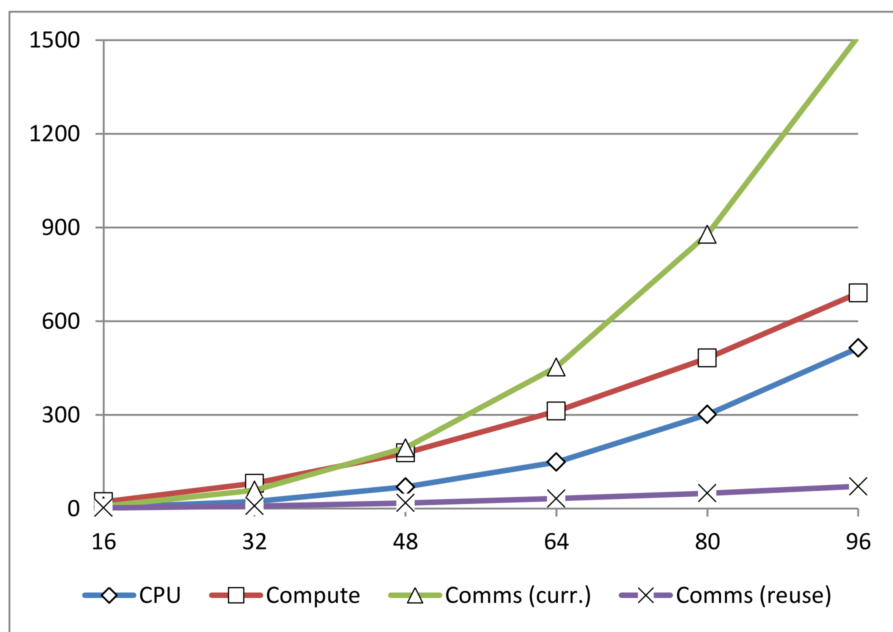
It can be observed that such a single memory design has a substantial negative impact on the achievable F_{max}, when compared to the original multiple-memories design. The best throughput is not achieved with the largest problem size. Moreover, the calculation performance is slower due to the bandwidth limitation of the single storage element, for example, a run-time difference of $27\mu s$ versus $1331\mu s$ for a problem size of 96. This clearly indicates the need for a non-trivial storage design and validates the sophisticated memory design process.

7.3.2 *Performance analysis*

The computational performance results for the accelerators are shown in Figure 50, comparing the hardware implementation with a pure software implementation running on the x86 CPU. The software code directly implements the initial algorithmic kernels, with no further optimisations. In these figures, 'Compute' and 'Comms' refer to a split of accelerator computation and communication (data transfer) times while the 'CPU' line is the complete execution time on the x86 CPU. Figure 50b includes two measurements for communication performance: the current ('curr.') implementation and a predicted implementation that employs data reuse ('reuse') techniques, the latter concept will be discussed further in Section 7.3.2.2. For realistic and fair comparisons, the channel between accelerator and the host sys-



(a) Stencil (parallel)



(b) Matrix factorisation

Figure 50: Execution time (primary axis, μs) of accelerator (at 200MHz, as 'Compute' & 'Comms') and x86 CPU ('CPU') over problem size.

tem has been simulated assuming an uncontested link with an aggregate bandwidth of 128-bits per clock cycle (3.05GiB/s), modelled as per Section 4.4. I make the worst-case comparative assumption that data communication on the CPU is cost free (i.e., not required). It is assumed that the CPU program and any required accelerator control schedules have been pre-loaded and so are not considered.

7.3.2.1 *Computational performance*

Figure 50 provides context around the performance prospects of the proposed design approach. It can be observed that overall computational performance ('Compute') of the accelerator designs are very good. Performance scales linearly for the stencil accelerator and, for problems larger than size 48, it out-performs the CPU. While the CPU remains faster for the matrix factorisation problem, the accelerator sees better performance scaling. Accelerator compute time scales linearly with problem size, which is reasonable because a sequential implementation will have a quadratic algorithmic complexity compared to the linear parallelised accelerator version. Overall, the results are very encouraging, particularly when considering the differences in clock frequencies and power usage. Moreover, since the accelerator architecture is non-pipelined there is untapped performance potential.

7.3.2.2 *Bandwidth and communication impact*

When including communication costs–between host and accelerator–the performance comparison needs to be adjusted to some extent. All experiments in Figure 50 have been simulated assuming a relatively high-speed 3.05 GiB/s channel (for example, realistic for PCIe 2.0 with 8 lanes). The communication time scales linearly with the available channel bandwidth. Clearly communication dominates the run-time performance of the stencil kernel at larger problem sizes, increasing quadratically, as does the run-time of the CPU implementation. The computation time of the accelerator only increases linearly because with larger problem sizes it also employs more PEs.

For the matrix factorisation case two communication costs are presented. The 'curr.' line shows the performance of the working implementation which does not reuse any data. Instead, within transitions of the outer time all data in the storage elements are completely transferred

backwards and forwards to the host CPU. This is an obvious limitation of the current design, and so the 'reuse' communication line shows the predicted cost when employing data reuse–transferring required data only. Data reuse has a particularly significant positive impact in this case; but it can only be applied to specific problems. This finding highlights a major area for future investigation.

While the computation performance of the accelerators is comparable with an x86 CPU (or even better in the stencil case), the required bandwidth is high in a scenario where data has to be transferred from the host CPU and back. However, it should be emphasised that the situation is different when the accelerators are paired with small (embedded) processors. In such situations there will not only be a power consumption advantage but also a strong performance advantage.

7.3.3 *Design-space exploration and usability*

Beyond performance there are other facets to any proposed development approach and tool-chains. Important concerns for adoption is whether the proposed approach offers any tangible benefits for developers to more rapidly create effective designs and what flexibility is on offer to explore different permutations of the design solution-space. Usability is above and beyond pure performance metrics, since poor usability means it is difficult to reach the potential performance highs.

In this work, it is shown that scattering functions can be used as a driver to achieve different design permutations based on simple mathematical constructions. Scattering functions describe the behavioural expectations of the design, and largely abstracts knowledge of implementation details from the designer. This is in contrast to other works, for example, the C2H Compiler (see Section 2.5), which optimise largely based on directives that specify low-level implementation details. Therefore, I believe the suggested approach facilitates an easy and methodical exploration of design-space.

Consider Equations 1 and 2: both scattering functions specify two very distinct design solutions for the stencil kernel, that resulted in wildly different resource utilisation and performance characteristics (see Figures 48a and 48b). For instance, for problem size 80 the throughput difference is huge–9.3 vs 314 MOP/s–but, so is resource usage–10 vs 800 DSP blocks.

Integrated in a professional development tool, the proposed concept can make it easier and faster for designers to develop accelerator designs. If a designer can describe the desired algorithmic behaviour with a scattering function then the tool-chain scan generate all necessary code without any further user intervention. Indicative predictions of resource usage and performance can also be computed and presented, without requiring a complete design synthesis through the entire flow, for fast evaluation of possible permutations.

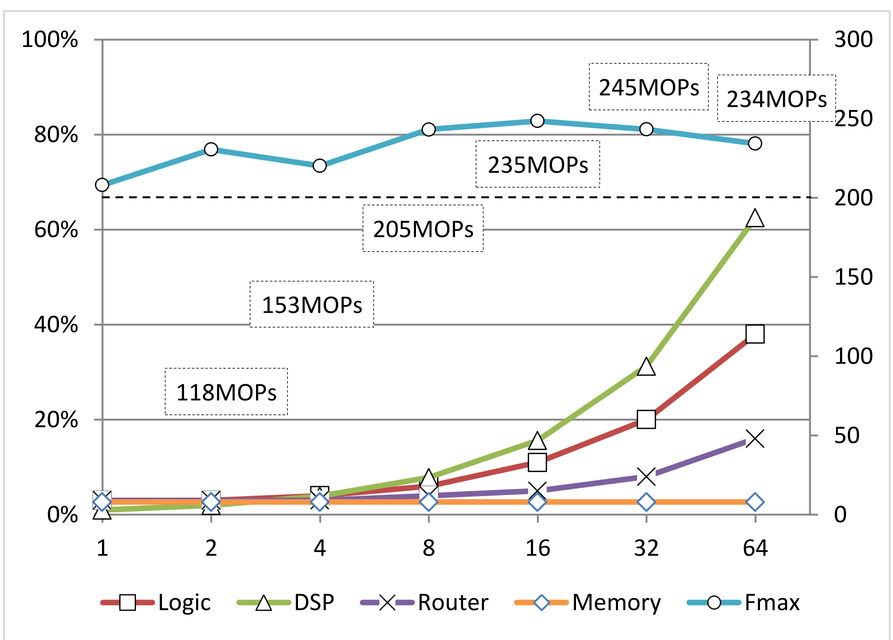
7.4 PIPELINED ACCELERATORS

The previous section presented very encouraging results of the experimental accelerators designs. However, with the addition of pipelining they can be improved even further. Pipelining is a key technique in hardware designs and is cornerstone to maximising throughput for a given amount of resources. Specifically, the accelerator control schedule is redesigned in order to make use of the pipelining within PEs. In other words, a single FP unit can be shared for independent calculations simply by feeding the data into its operating pipeline. Chapter 4.6 discussed this process in greater detail. The vendor provided floating-point cores that are used for the PE design are inherently pipelined, so this is a very beneficial technique.

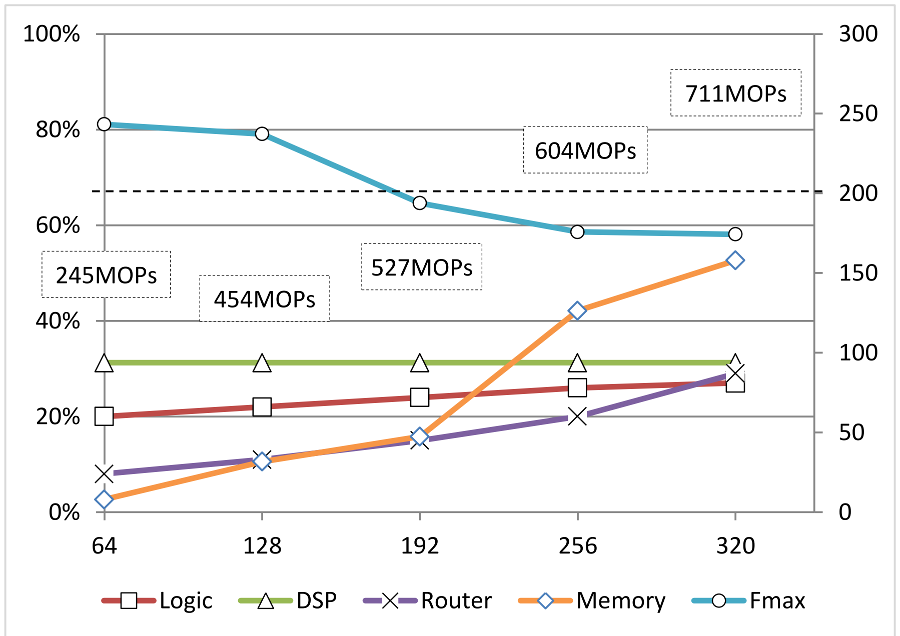
This Section presents results from accelerator designs which employ pipelining to different degrees. I explore this technique from two consideration points: 1) examining the effect of pipelining to solve a fixed-size problem, and 2) examining how a fixed-size accelerator (number of PEs) can be scaled to solve larger problem sizes. To recap, the current implementation of pipelining creates a memory layout with respect to problem size (as if it were a totally unconstrained design) but pairs this with a multiplexer array to modulate and share connections to the reduced number of physical PEs.

7.4.1 *Increased accelerator efficiency*

To evaluate the first point, a problem size of 64 has been chosen. Pipelining is applied such that the number of physical PEs are reduced, yet the pipeline utilisation is increased—and, by reciprocal, the number of operations performed within a logical tick. Figure 51a captures the synthesis results for this scenario.



(a) Stencil kernel with problem size of 64, and number of PEs scaled from 1 to 64 (where pipeline utilisation = $64/n_{PE}$).



(b) Stencil kernel with 32 PEs to address problem size scaled from 64 to 320 (where pipeline utilisation = $n_{problem}/32$).

Figure 51: Synthesis results of pipelined HDL accelerator designs.

The results show that the resource usage scales proportionally with the number of physical processors available. Moreover, the achieved target F_{max} is relatively stable across the board. That said, it is evident that the frequency generally trends downward as the pipeline utilisation increases. This is a reasonable side-effect since the memory-PE multiplexer network becomes larger and starts to have an impact. It is important to note that the pipeline depth is fixed across all experiments at 25 stages; the PE core is fully pipelined and the depth is determined by the number of stages in the floating-point operations comprising the critical path.

Looking at the MOP/s throughput for all the configurations it can be seen that pipelining has a profound effect. Despite having only half and even a quarter of the number of physical processors, the pipelined designs achieved greater overall throughput (245 and 235 MOP/s). However, throughput falls off from 4 PEs (one eighth of the unconstrained processor count) as the extra latency of each logical tick (the delay to wait for the pipeline to be completely emptied) and decrease in clock frequency has a larger impact. This is due to the fact that computation and communication do not overlap in the current approach, but it could be improved later.

This shows that it is relatively easy to realise the same performance with considerably lower resource usage by applying pipelining, making it an ideal technique to improve the efficiency (throughput per resource consumption). Pipelining makes more parts of the FPGA work simultaneously, which is the design goal to achieve high efficiency and thereby high throughput.

7.4.2 *Targeting large problems*

Conversely, we can apply pipelining as a mechanism for design scalability; specifically, to accelerate larger problem sizes. Experiments have been conducted which fix the number of available physical PEs to 32 and then scale the problem size from 64 all the way to 320. In the process, the pipeline utilisation is dynamically adjusted from 2–10, respectively. Figure 53 captures the synthesis results for this scenario.

Immediately, the potential for achieving significant throughput is shown–over 700MOP/s for the largest problem size–while using only one third of the available DSP elements. Unsurprisingly, the clock frequency has fallen below the target; most likely as a result of the

more complex multiplexing necessary in this design. To put this into a concrete performance context: this works out to a computation-only time of only 145μs (with communication time of 592μs) versus 560μs total time on the Intel x86 workstation.

7.4.3 *Limitations to scalability*

Unfortunately, however, the results indicate that the architecture approaches other limits in terms of memory availability and decreasing clock frequencies. For the largest problem size in Figure 53, memory consumption is 53% of total available bits but, in fact, all 1280 M9K memory blocks have been used in order to allocate this space. This is a direct outcome of creating memories with respect to problem size; therefore, more of these smaller M9K blocks are required to create fully-functional independent memories.

Another thing to note, is that there is a jump in memory consumption between the 192 and 256 problem sizes which is due to increasing the size of the control program memory in order to handle the larger problem. This leads to the RAM address width becoming expanded by one bit, doubling required capacity, in order to fit the control program.

7.4.3.1 *Mitigating larger resource-sharing multiplexer networks*

Conceptually, pipelining is extremely effective for increasing accelerator efficiency and can be seen as a "no brainer". But, it has been shown that the proposed implementation it is not practically feasible for very large pipeline loads and problem sizes.

Analysis of these findings identifies that the biggest contributing factor is the memory allocation policy. When employing pipelining, there are more dedicated memories associated with fewer virtual processors, where the pipeline utilisation represents the scale of this disparity. This is not directly caused by the use of pipelining, but, rather, it is a side-effect of supporting so many independent memories–which positively contributes to the impressive throughput. Moreover, the mapping leads to the creation of large multiplexer networks which also affects routing and the achievable F_{max}.

Some of these limitations can be mitigated with modifications to the design of the accelerator memory allocation, drawing inspiration from the multiple PEs, single RAM concept experiment (Figure

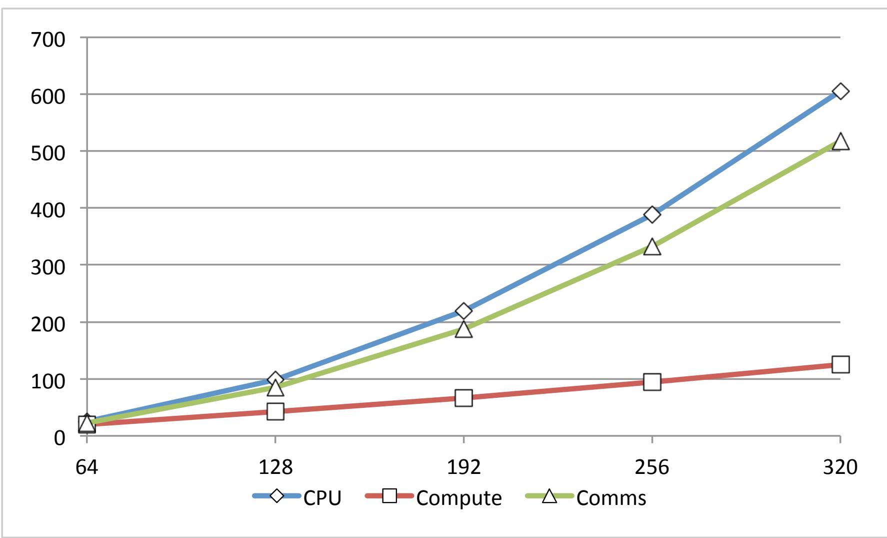


Figure 52: Execution time (primary axis, μs) of pipelined stencil accelerator (at 200MHz, as 'Compute' & 'Comms') and x86 CPU ('CPU') over problem size.

49a). This entails understanding of the trade-off between memory bandwidth (with respect to the availability of distinct memory blocks for usable RAM ports) and throughput for a kernel. Future designs should not create only as many memories as there are physical processors and, instead, generically partition this into a virtual address space to create 'logical memories'. The mapping of logical to physical memories could be a tuneable parameter to refine the overall performance of the accelerator, and can likely be automated. This could bring an incremental improvement to F_{max} performance, particularly if paired with further increases in device size.

Furthermore, a more radical approach would be to explore the opportunities for different data layouts and localised data reuse opportunities to maximise bandwidth for any given number of memories. For example, the stencil kernel example, presented in this work, has a natural data reuse opportunity where a recently written stencil data item is read again for the next, adjacent, computation. In-flight data caching can be used to buffer data in both read and write paths.

7.4.4 *Performance analysis*

The experiments of 7.3.2 were reflected with fully-pipelined processing units, to examine the impact on performance. In this case, larger prob-

lems are considered, which are possible thanks to the resource sharing attribute of pipelining.

Figure 52 shows a performance comparison between the accelerator design for different problem sizes, where the number of processors is fixed at 32. Therefore, as the problem size increases, so does the pipeline utilisation. As expected, the computation performance is very impressive, especially at higher pipeline utilisation, where it has a significant advantage over the 2.4 GHz x86 CPU. But, the data communication time still has a significant bearing on the overall performance.

7.5 TILED PROBLEM SPACE

Remember that tiling is applied to the target polyhedron (execution schedule) and not the iteration space.

Tiling is used to break the problem space into smaller pieces such that a larger problem can be solved using a fixed amount of resources. To put it in context of this research, it is necessary to employ tiling such that arbitrarily large problems can be accelerated. Primarily, tiling requires careful manipulation of the control structures that specify kernel execution, along with some minor architectural modifications to handle the changes to the control program and memory layout as a function of tile size rather than the problem size.

Therefore, tiling provides a robust mechanism for scalability to address any input kernel size. Moreover, multiple tile 'units' (instances of the tiling-capable hardware module) can be used concurrently within a larger multi-unit accelerator design. This multi-unit approach can improve throughput, by exploiting any available coarse-grained concurrency in the schedule. It is also more amenable to synthesis because, typically, tile sizes (and corresponding hardware units) are individually smaller than for a single monolithic, unconstrained, accelerator.

Pipelining and tiling can also be combined: each tile hardware unit can employ pipelining for greater efficiency. However, this section presents only non-pipelined results.

7.5.1 *Validation of synthesis characteristics*

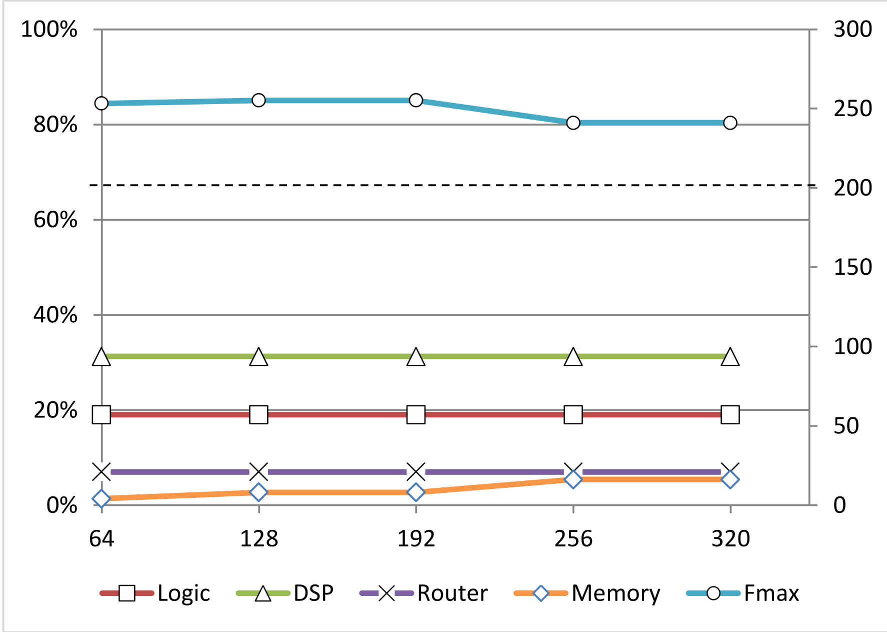


Figure 53: Synthesis results for tiled stencil kernel (32x512 configuration) to address problem size scaled from 64 to 320.

This experiment validates that applying tiling should not have a significant impact on the synthesis characteristics of an accelerator, the results are shown in Figure 53. Once again, problem size has been scaled from 64 to 320, but the tile configuration has been kept constant at 32-wide by 512-deep (that is, 32 PEs with a control program capacity of 512 time-steps). This configuration relates to the way in which the target polyhedron is split which, practically, has a bearing on the resources within a tile unit. The width corresponds directly to the number of physical PEs available and so in this case every design has 32 PEs. The depth of the tile corresponds to the number of logical ticks that can be processed which in turn directs the maximum capacity of each memory within the unit.

Refer to Section 5.4 for greater detail about tiling and how it is applied.

The presented synthesis results agree with the premise that tiling should have little bearing on synthesis results. One point to note is the small jump in memory usage between problem size 192 and 256. This is due to the size of the control schedule RAM doubling to accommodate a larger schedule. The code generation process automatically sizes this RAM based on the input problem size. All other metrics show consistency across all the presented problem sizes, which is the

expected behaviour. Moreover, the results align closely with the problem size 32 result from Figure 48a—a very similar design, but without tiling support.

7.5.2 *Multi-unit scalability*

Tiling provides another processing dimension that can be used to employ coarse grained concurrency, if desired. This requires multiple hardware units. All data synchronisation is performed by the host system control program, which is the authoritative memory source, so data necessary for each tile computation must be communicated before and after each computation. It is possible to have multiple tile units within a single FPGA and is equally viable to distribute multiple units across separate FPGA devices. In some circumstances, the latter may provide greater overall system bandwidth if the devices are not connected to a shared data bus.

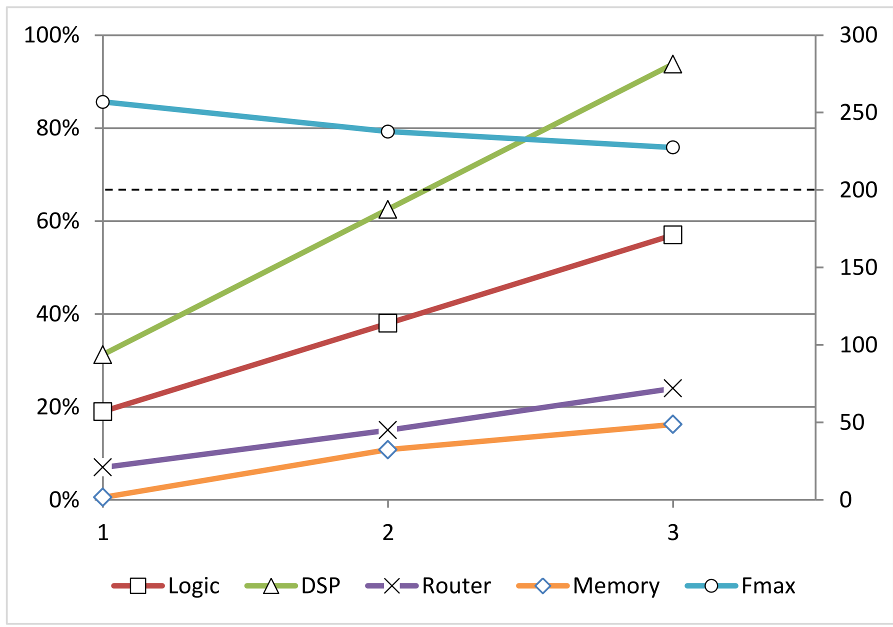


Figure 54: Synthesis results for multiple (1 to 3) tiled units (32x512 configuration) to address problem size 320x320.

While the benefits of tile-level concurrency has not been deeply explored in this work, an experiment has been conducted to verify that multiple units on a single device can be a viable option with

respect to synthesis scalability. Figure 54 depicts just that, with multiple tile units for the problem size 320, with a 32x512 tile configuration synthesised for a single device. The scalability is near linear and the 200 MHz F_{max} target has been met across the board. It should be stressed that because of the smaller, simple architecture, the three-unit design achieves a higher clock frequency compared to the problem size 96x96 result from Figure 48a, despite both using the same amount of PEs.

Practically, these results also indicate how three different accelerator tile units (for three different kernels) can easily co-exist within the same device. It adds another dimension to the design-space, since each unit can be configured separately according to different performance and resource cost objectives. A multi-unit approach may also allow for more efficient utilisation of available host interconnect bandwidth, as some tile units can be processing, while another receives or sends data.

8 CONCLUSIONS

FPGAs offer immense potential to accelerate computing applications within a reconfigurable computing environment. They provide designers a blank canvas from which to sculpt a bespoke hardware architecture and parallel datapath, complete with enormous, distributed, bandwidth from a custom memory topology and hierarchy. This can be applied to demanding applications across the scientific and business domains alike.

However, realising and delivering this potential performance in the real-world represents a significant challenge that has not yet been adequately addressed. Research in this area has focused on a diverse range of approaches, with the two most pertinent being: languages and tools to aid in hardware and system design; and compilation and mapping techniques to target application code to generated FPGA hardware. The latter is especially interesting, from the perspective of rapid adoption and embracement from the mainstream development community, because it creates a pathway from which the large body of existing codes can be accelerated at little cost.

This thesis has investigated and advanced this from two facets: 1) the acceleration of high-level code by integrating hardware data structures, and 2) using the polyhedral model for generating accelerators for numeric kernels, based on succinct mathematical scattering functions to describe the transformation. While the two ideas may initially seem adjunct, they naturally cut across the range of applications that would benefit from acceleration.

ACCELERATED DATA STRUCTURES AND HW/SW INTEGRATION

An initial and motivating observation in this thesis was that some data structures are very amenable to hardware implementation. Consequently, this thesis investigated the hardware acceleration of such data structures by focusing on the priority queue (PQ). A pure hardware implementation has superior performance characteristics, with constant time complexity obtained for both insertion and extraction operations compared to logarithmic time insertion for the best software implementation, albeit at a limited size. But, the interesting challenge for this approach is how this performance can be made both accessible and scalable for application developers.

Using the Java language as a case study, a new priority queue API was proposed to automatically leverage a hardware accelerator, if available. Java's native interface layer was employed within which the platform-specific hardware system calls can be implemented, without impacting the high-level programming model. This made communication between the application and the hardware accelerator straightforward and effective.

The proposed hardware accelerator, while very fast, was limited in scalability by available hardware resources. Conversely, the software implementations are slow, but scalability was easy, and limited only by memory capacity. Solving the scalability challenge was crucial for the viability and usefulness of the proposed approach. The solution was to combine and harness the advantages of both approaches using hardware/software co-design techniques to leverage both aspects of the reconfigurable system.

This led to the design and development of the hybrid PQ. It used the hardware accelerator combined with a software PQ to extend the capacity by capturing over-flow when the hardware queue was full. This was achieved by peeking at the low-priority end of the hardware queue in order to decide whether to write to hardware, software, or both; depending on the situation.

Overall, the hybrid PQ demonstrated the merit of this approach and delivered speedups (1.5–3x) over the pure-software implementation in a realistic scenario computing the minimum spanning tree using Prim's algorithm. It makes for a particularly attractive solution when the data sizes are small enough to mostly reside on the FPGA

or the host system is relatively weak, for example, in embedded systems.

Overall, this research validated the promise and benefit of hardware accelerated data structures. The proposed approach allows software engineers to use hardware accelerated data structures in a high-level language such as Java gaining significant performance benefits.

Future research avenues

There are several unique research angles that can extend the work in this thesis. These include:

- Investigating and evaluating alternative hardware PQ architectures which may offer better scalability, for example, a binary heap implementation, and evaluate the performance trade-off.
- Extending the hybrid approach to hardware, where both the queue parts, the fast and the scalable, are realised in hardware. This is also interesting in combination with the previous point. Research on this area is already undertaken at the University of Auckland in the Parallel and Reconfigurable Computing group.
- Investigate other data structures and application areas that can benefit from hardware acceleration.
- Examine the hardware/software interface in greater depth to determine if there are opportunities to improve the coupling between the application and the accelerator, whilst removing sources of latency.

POLYHEDRAL-ASSISTED ACCELERATOR GENERATION

This work started from the premise that the polyhedral model approach would be an effective way to specify and generate accelerators for numeric kernels. Moreover, minimal modification is required to existing application code; avoiding the need for large rewrites of already working applications to fit with alternative programming paradigms, or learning domain specific languages.

The foundation of this approach is the architectural model that was proposed for the accelerator. Examining the composition of contemporary FPGAs, it was found that the memory topology is a signific-

ant advantage that can be used to provide extraordinary distributed bandwidth, when compared to competing technologies, because of the sheer number of independent memories. Furthermore, registers provide fast, random-access read and write storage. The novel approach of this thesis uses these observations by directly connecting storage elements to processing elements (small, computational cores) as needed. The proposed interconnection is based on a simple point-to-point model, where multiplexers are used to arbiter multiple input drivers. Given the available routing capacity on modern FPGA devices, I theorised that the available logic would likely be exhausted before the network performance declined below acceptable levels. The experimental results supported this argument.

The flexibility and simplicity of the novel architectural model is instrumental to effectively map algorithmic kernels to semi-automatically generated hardware. In the proposed approach, using the polyhedral model, mapping begins with the user providing a mathematical scattering function that describes the behaviour transformation from algorithm to accelerator. Using the polyhedral model means to apply this function to all statement instances within the input kernel (the algorithm to accelerate) so that they are mapped to an output polyhedron with temporal and spatial dimensions. In other words, the output is a, usually parallel, execution schedule of the kernel. These semantics are extremely amenable to hardware generation since they can form the basis of the complete control program for a hardware accelerator.

An experimental toolchain, polyAcc, has been developed to use this output polyhedron in-conjunction with the datapath information to generate a full control unit program that implements the kernel. The scattering function paired with the dependence and access functions of the input kernel itself are additional inputs for the generation of the datapath. A novel and innovative aspect of the proposed approach is the automatic creation of a storage architecture using the polyhedral information. To provide the highest throughput, by using the available memory bandwidth, a dot product test has been devised to correlate the memory accesses with the scattering function in order to select the performance-optimised storage architecture. The success of polyAcc is strong evidence that a completely automated development flow is realistic and achievable, given a scattering function.

Experiments were conducted for two common constructions: a stencil kernel and matrix factorisation. Moreover, different configurations were tested from a typical parallel implementation to sequential. Overall the performance proved to be very competitive and scaled clearly with hardware usage. Moreover, the impact of the proposed storage mapping proved beneficial when compared to a manual single memory implementation. When compared to a workstation CPU, the FPGA computational performances scaled linearly and surpassed it at large problems but, due to the high communication costs, it was not faster overall.

The new semi-automatic approach was further improved by integrating pipelining to improve the efficiency of the accelerator, for very little cost. Multiple sets of data are multiplexed to a single processing element, while pipelining improves throughput significantly. Experimental results proved that it had an impact on the design space, for example a 64 PE design improved from 234 to over 700 MOP/s throughput. However, the ability to exploit pipelining in the proposed architecture is curtailed by the exhaustion of memory devices (or, bandwidth) to provide and support the flow of non-dependent data sets.

Scalability of the approach has been addressed by leveraging tiling to partition large problems. The smaller pieces can run on tiled accelerator units that are adequately sized to fit within a device. Furthermore, tiling supports coarse-grained concurrency such that multiple such units could be present on a device, mitigating any potential routing network scalability issues, and many units could be employed in a cluster. Unlike conventional loop tiling, this technique focuses on tiling the target polyhedron and not the iteration space. Since the target polyhedron already maintains guarantees of legality, it is easy to reason about the legality of partitioning. Experimental results show that incorporating tiling has little overhead on the synthesis results for an accelerator.

To conclude, this thesis has presented a complete methodology to construct a hardware accelerator for numeric loop kernels, from the input algorithm and succinct scattering functions. Using a semi-automatic approach, design-space exploration is dramatically improved because the output accelerator behaves as the scattering function describes, making it easy to design for parallelism. Moreover, the experimental results have demonstrated high correlation between expected

and actual performance, with little overhead from the proposed architectural target model.

Future research avenues

The author believes the polyhedral model holds potential for future research into application-synthesis for accelerator generation. Key areas of investigation include:

- Automatic searching of scattering functions based on simple heuristics. The legality and performance of scattering functions can be tested relatively easily. Typically, many of the same optimisations are used to extract parallelism from loop nests; these can be catalogued and applied, automatically.
- Auto-tuning provides opportunity for the exploration of the design-space. This is particularly applicable to scattering function agnostic tuning that can not be easily predicted, for example, using pipelining and tiling.
- A more radical memory selection approach can be investigated, which explores the opportunities for different data layouts and localised data reuse. This should maximise bandwidth for any given amount of memories. Integer programming formulations can be used as a vehicle for optimisation.
- The target architecture can be extended to employ caching buffers, a natural mechanism for data reuse. For example, with the stencil kernel a recently written stencil data item is read again for the next, adjacent, computation. In-flight data can be cached and provide a hybrid systolic data flow.
- Overlapped communication and computation has not been considered in this thesis. This could be explored to propose a generic way to handle high-bandwidth data exchange.

BIBLIOGRAPHY

[1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: principles, techniques, & tools*, 2nd ed. Pearson/Addison Wesley, 2007. (Cited on page 105.)

[2] Alias, C., Pasca, B., and Plesco, A. Automatic generation of FPGA-specific pipelined accelerators. In *Reconfigurable Computing: Architectures, Tools and Applications*, A. Koch, R. Krishnamurthy, J. McAllister, R. Woods, and T. El-Ghazawi, Eds., vol. 6578 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2011, pp. 53–66. (Cited on page 32.)

[3] Allen, R., and Kennedy, K. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann, San Francisco, CA, USA, 2001. (Cited on page 83.)

[4] Altera Corporation. Automated generation of hardware accelerators with direct memory access from ANSI/ISO standard C functions. Tech. rep., May 2006. (Cited on page 23.)

[5] Altera Corporation. AN456: PCI Express high performance reference design. Tech. rep., August 2012. Version 1.5. (Cited on page 71.)

[6] Altera Corporation. *DSP Builder Handbook*, 2012. (Cited on page 24.)

[7] Altera Corporation. OpenCL for Altera FPGAs: Accelerating performance and design productivity. http://www.altera.com/products/software/opencl/opencl-index.html, 2013. (Cited on page 24.)

[8] Altera Corporation. Stratix IV FPGA family architecture. http://www.altera.com/devices/fpga/stratix-fpgas/stratix-iv/overview/architecture/stxiv-architecture.html, 2013. (Cited on page 62.)

[9] Altera Corportation. *Nios II C2H Compiler: User Guide*, November 2009. (Cited on pages 25 and 26.)

[10] Asano, S., Maruyama, T., and Yamaguchi, Y. Performance comparison of FPGA, GPU and CPU in image processing. In *Proc. of the Intl. Conf. on Field Programmable Logic and Applications (FPL'09)* (2009), pp. 126–131. (Cited on page 4.)

[11] Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., and Asanović, K. Chisel: constructing hardware in a Scala embedded language. In *Proc. of the Annual Design Automation Conference (DAC'12)* (2012), ACM, pp. 1216–1225. (Cited on page 20.)

[12] Banerjee, U., Eigenmann, R., Nicolau, A., and Padua, D. A. Automatic program parallelization. *Proc. of the IEEE 81*, 2 (February 1993), 211–243. (Cited on page 83.)

[13] Bastoul, C. Code generation in the polyhedral model is easier than you think. In *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'04)* (2004), IEEE Computer Society, pp. 7–16. (Cited on pages 86, 87, 117, 120, and 136.)

[14] Bastoul, C., Cohen, A., Girbal, S., Sharma, S., and Temam, O. Putting polyhedral loop transformations to work. In *Languages and Compilers for Parallel Computing*, L. Rauchwerger, Ed., vol. 2958 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2004, pp. 209–225. (Cited on pages 82 and 100.)

[15] Bayliss, S., and Constantinides, G. A. Optimizing SDRAM bandwidth for custom FPGA loop accelerators. In *Proc. of the Intl. Symp. on Field Programmable Gate Arrays (FPGA'12)* (2012), ACM, pp. 195–204. (Cited on page 33.)

[16] Bellows, P., and Hutchings, B. JHDL: an HDL for reconfigurable systems. In *Proc. of the Symp. on FPGAs for Custom Computing Machines (FCCM'98)* (1998), pp. 175–184. (Cited on page 18.)

[17] Bergeron, E., Saint-Mleux, X., Feeley, M., and David, J. P. High level synthesis for data-driven applications. In *Proc. of the Intl. Workshop on Rapid System Prototyping (RSP'05)* (2005), pp. 54–60. (Cited on page 19.)

[18] Bhagwan, R., and Lin, B. Fast and scalable priority queue architecture for high-speed network switches. In *Proc. of the Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'00)* (2000), vol. 2, pp. 538–547. (Cited on page 40.)

[19] Bloom, G. *Operating System Support for Shared Hardware Data Structures*. PhD thesis, The George Washington University, 2013. (Cited on page 56.)

[20] Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. A practical automatic polyhedral parallelizer and locality optimizer. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'08)* (2008), ACM, pp. 101–113. (Cited on pages 32 and 34.)

[21] Boussinot, F., and de Simone, R. The esterel language. *Proceedings of the IEEE 79*, 9 (1991), 1293–1304. (Cited on page 21.)

[22] Buell, D., El-Ghazawi, T., Gaj, K., and Kindratenko, V. Guest Editors' Introduction: High-performance reconfigurable computing. *Computer 40*, 3 (March 2007), 23–27. (Cited on page 15.)

[23] Cadence Design Systems. C-to-silicon compiler: High-level synthesis. Datasheet, 2008. (Cited on page 21.)

[24] Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J. H., Brown, S., and Czajkowski, T. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proc. of the Intl. Symp. on Field Programmable Gate Arrays (FPGA'11)* (2011), ACM, pp. 33–36. (Cited on page 24.)

[25] Cardoso, J. M. P., and Neto, H. C. Compilation for FPGA-based reconfigurable hardware. *IEEE Design & Test of Computers 20*, 2 (2003), 65–75. (Cited on page 25.)

[26] Casanova, H., Legrand, A., and Robert, Y. *Parallel Algorithms*. Chapman & Hall, 2009. (Cited on pages 64 and 86.)

[27] Chandra, R., and Sinnen, O. Improving application performance with hardware data structures. In *Proc. of the Intl. Symp. on Parallel & Distributed Processing, Workshops and PhD Forum (IPDPSW'10)* (2010), IEEE, pp. 1–4. (Cited on page 56.)

[28] Chandra, R., and Sinnen, O. Towards automated optimisation of tool-generated HW/SW SoPC designs. In *Proc. of the Intl. Symp. on Field Programmable Gate Arrays (FPGA'11)* (2011), ACM, pp. 285–285. Abstract only. (Cited on page 31.)

[29] Chang, C., Wawrzynek, J., and Brodersen, R. W. BEE2: a high-end reconfigurable computing system. *IEEE Design & Test of Computers 22*, 2 (2005), 114–125. (Cited on page 11.)

[30] Chao, H. J. A novel architecture for queue management in the ATM network. *IEEE Journal on Selected Areas in Communications 9*, 7 (1991), 1110–1118. (Cited on page 40.)

[31] Chu, P. P. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. Wiley-IEEE Press, 2006. (Cited on page 76.)

[32] Compton, K., and Hauck, S. Reconfigurable computing: a survey of systems and software. *ACM Computing Surverys 34*, 2 (2002), 171–210. (Cited on page 4.)

[33] Cong, J., Jiang, W., Liu, B., and Zou, Y. Automatic memory partitioning and scheduling for throughput and power optimization. *ACM Trans. Des. Autom. Electron. Syst. 16*, 2 (Apr. 2011), 15:1–15:25. (Cited on page 33.)

[34] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*, 2nd ed. The MIT Press, 2001. (Cited on pages 37 and 39.)

[35] Coussy, P., and Morawiec, A., Eds. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer Netherlands, 2008. (Cited on page 29.)

[36] DeHon, A. Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% LUT utilization). In *Proc. of the Intl. Symp. on Field Programmable Gate Arrays (FPGA'99)* (1999), ACM, pp. 69–78. (Cited on page 70.)

[37] Devos, H., Beyls, K., Christiaens, M., Van Campenhout, J., DH́ollander, E., and Stroobandt, D. Finding and applying loop transformations for generating optimized FPGA implementations. In *Transactions on High-Performance Embedded Architectures and Compilers I*, P. Stenström, Ed., vol. 4050 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2007, pp. 159–178. (Cited on page 32.)

[38] Flynn, M. J. Very high-speed computing systems. *Proc. of the IEEE 54* (1966), 1901–1909. (Cited on page 13.)

[39] Forte Design Systems. Cynthesizer: The most productive path to silicon. Brochure, 2008. (Cited on page 21.)

[40] Galloway, D. The Transmogrifier C hardware description language and compiler for FPGAs. In *Proc. of the Symp. on FPGAs for Custom Computing Machines (FCCM'95)* (1995), pp. 136–144. (Cited on page 18.)

[41] Geer, D. Chip makers turn to multicore processors. *Computer 38*, 5 (May 2005), 11–13. (Cited on page 1.)

[42] Gibbons, A. *Algorithmic Graph Theory*. Cambridge University Press, 1985. (Cited on pages 37 and 38.)

[43] Girbal, S., Vasilache, N., Bastoul, C., Cohen, A., Parello, D., Sigler, M., and Temam, O. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Program. 34* (June 2006), 261–317. (Cited on page 81.)

[44] Gokhale, M., and Graham, P. S. Reconfigurable computing systems. In *Reconfigurable Computing*. Springer US, 2005, pp. 37–50. (Cited on page 10.)

[45] Grama, A., Gupta, A., Karypis, G., and Kumar, V. *Introduction to Parallel Computing*, 2nd ed. Addison-Wesley, New York, 2003. (Cited on page 39.)

[46] Grosser, T., Zheng, H., Aloor, R., Simbürger, A., Grösslinger, A., and Pouchet, L.-N. Polly: Polyhedral optimization in LLVM. In *Proc. of the Intl. Workshop on Polyhedral Compilation Techniques (IMPACT)* (2011). (Cited on pages 32 and 136.)

[47] Gruian, F., Roop, P., Salcic, Z., and Radojevic, I. The SystemJ approach to system-level design. In *Proc. of the Intl. Conf. on Formal Methods and Models for Co-Design (MEMOCODE'06)* (2006), pp. 149–158. (Cited on page 21.)

[48] Guo, Z., Buyukkurt, B., Cortes, J., Mitra, A., and Najjar, W. A compiler intermediate representation for reconfigurable fabrics. *Int. J. Parallel Program. 36* (October 2008), 493–520. (Cited on page 23.)

[49] Hammes, J., Rinker, B., Bohm, W., Najjar, W., Draper, B., and Beveridge, R. Cameron: high level language compilation for reconfigurable systems. In *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'99)* (1999), pp. 236–244. (Cited on page 19.)

[50] Hannig, F., Ruckdeschel, H., Dutta, H., and Teich, J. PARO: Synthesis of hardware accelerators for multi-dimensional dataflow-intensive applications. In *Reconfigurable Computing: Architectures, Tools and Applications* (2008), vol. 4943 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 287–293. (Cited on pages 19 and 32.)

[51] Hartenstein, R. A decade of reconfigurable computing: a visionary retrospective. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE'01)* (2001), pp. 642–649. (Cited on pages 4 and 15.)

[52] Hoffman, A., Meyr, H., and Leupers, R. *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, 2002. (Cited on page 20.)

[53] Impulse Accelerated Technologies Inc. Accelerate C in FPGA. Brochure, 2007. (Cited on page 23.)

[54] Jääskeläinen, P., de La Lama, C., Huerta, P., and Takala, J. OpenCL-based design methodology for application-specific processors. In *Intl. Conf. on Embedded Computer Systems (SAMOS)* (2010), pp. 223–230. (Cited on page 24.)

[55] Jones, D., Powell, A., Bouganis, C., and Cheung, P. Y. K. GPU versus FPGA for high productivity computing. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on* (2010), pp. 119–124. (Cited on page 4.)

[56] Kapre, N., and DeHon, A. Performance comparison of single-precision SPICE model-evaluation on FPGA, GPU, Cell, and multi-core processors. In *Proc. of the Intl. Conf. on Field Programmable Logic and Applications (FPL'09)* (2009), pp. 65–72. (Cited on page 4.)

[57] Karimi, K., Dickson, N. G., and Hamze, F. A performance comparison of CUDA and OpenCL. *CoRR abs/1005.2581* (2010). (Cited on page 3.)

[58] Karp, R. M., Miller, R. E., and Winograd, S. The organization of computations for uniform recurrence equations. *J. ACM 14* (July 1967), 563–590. (Cited on pages 32 and 81.)

[59] Kennedy, K., and Allen, J. R. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. (Cited on page 28.)

[60] Kindratenko, V. V., Steffen, C. P., and Brunner, R. J. Accelerating scientific applications with reconfigurable computing: Getting started. *Computing in Science & Engineering 9*, 5 (2007), 70–77. (Cited on page 11.)

[61] Kung, S. Y. *VLSI array processors*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987. (Cited on pages 32 and 81.)

[62] Lamport, L. The parallel execution of DO loops. *Commun. ACM 17* (February 1974), 83–93. (Cited on pages 32 and 81.)

[63] Lattner, C., and Adve, V. LLVM: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on* (2004), pp. 75–86. (Cited on page 24.)

[64] Lau, D., Pritchard, O., and Molson, P. Automated generation of hardware accelerators with direct memory access from ANSI/ISO standard C functions. In *Proc. of the Intl. Symp. on Field-Programmable Custom Computing Machines (FCCM'06)* (apr. 2006), pp. 45 –56. (Cited on page 31.)

[65] Lee, H. G., Chang, N., Ogras, U. Y., and Marculescu, R. On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip ap-

proaches. *ACM Trans. Des. Autom. Electron. Syst. 12*, 3 (May 2008), 23:1–23:20. (Cited on page 96.)

[66] Lengauer, C. Loop parallelization in the polytope model. In *Proc. of the Intl. Conf. on Concurrency Theory (CONCUR'93)* (1993), Springer-Verlag, pp. 398–416. (Cited on page 82.)

[67] Liang, S. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, 1999. (Cited on page 46.)

[68] Liu, Q., Constantinides, G., Masselos, K., and Cheung, P. Automatic on-chip memory minimization for data reuse. In *Proc. of the Intl. Symp. on Field-Programmable Custom Computing Machines (FCCM'07)* (2007), pp. 251–260. (Cited on page 33.)

[69] Marcus, D. A. *Graph Theory: A Problem Oriented Approach*. The Mathematical Association of America, 2008. (Cited on page 38.)

[70] Mentor Graphics Corp. Catapult C synthesis. Datasheet, 2008. (Cited on page 18.)

[71] Mishra, P., and Dutt, N., Eds. *Processor Description Languages: Applications and Methodologies*, vol. 1 of *Systems on Silicon*. Morgan Kaufmann Publishers/Elsevier, 2008. (Cited on page 20.)

[72] Mittal, G., Zaretsky, D., Memik, G., and Banerjee, P. Automatic extraction of function bodies from software binaries. In *Proc. Asia and South Pacific Design Automation Conference the ASP-DAC 2005* (2005), vol. 2, pp. 928–931 Vol. 2. (Cited on page 25.)

[73] Möhl, S. The Mitrion-C programming language. Manual, 2006. (Cited on page 23.)

[74] Moon, S.-W., Rexford, J., and Shin, K. G. Scalable hardware priority queue architectures for high-speed packet switches. *IEEE Transactions on Computers 49*, 11 (2000), 1215–1227. (Cited on pages 40 and 41.)

[75] Moore, G. E. Cramming more components onto integrated circuits. *Proceedings of the IEEE 86*, 1 (1998), 82–85. (Cited on page 1.)

[76] Oruklu, E., Hanley, R., Aslan, S., Desmouliers, C., Vallina, F. M., and Saniie, J. System-on-chip design using high-level

synthesis tools. *Circuits and Systems 3*, 1 (2012), 1–9. (Cited on page 30.)

[77] Papadonikolakis, M., Bouganis, C., and Constantinides, G. Performance comparison of GPU and FPGA architectures for the SVM training problem. In *Proc. of the Intl. Conf. on Field-Programmable Technology (FPT'09)* (2009), pp. 388–391. (Cited on page 4.)

[78] Pham, D., Asano, S., Bolliger, M., Day, M., Hofstee, H., Johns, C., Kahle, J., Kameyama, A., Keaty, J., Masubuchi, Y., Riley, M., Shippy, D., Stasiak, D., Suzuoki, M., Wang, M., Warnock, J., Weitzel, S., Wendel, D., Yamazaki, T., and Yazawa, K. The design and implementation of a first-generation CELL processor. In *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International* (2005), pp. 184–592 Vol. 1. (Cited on page 2.)

[79] Pouchet, L.-N., Bastoul, C., Cohen, A., and Vasilache, N. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO'07)* (2007), IEEE Computer Society, pp. 144–156. (Cited on page 34.)

[80] Pouchet, L.-N., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., and Sadayappan, P. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Proc. of the Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'10)* (2010), IEEE Computer Society, pp. 1–11. (Cited on page 34.)

[81] Quinton, P. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proceedings of the 11th annual international symposium on Computer architecture* (New York, NY, USA, 1984), ISCA '84, ACM, pp. 208–214. (Cited on page 81.)

[82] Rao, S. K., and Kailath, T. Regular iterative algorithms and their implementation on processor arrays. *Proceedings of the IEEE 76*, 3 (mar 1988), 259 –269. (Cited on pages 32 and 81.)

[83] Rexford, J., Hall, J., and Shin, K. G. A router architecture for real-time communication in multicomputer networks. *IEEE*

Transactions on Computers 47, 10 (1998), 1088–1101. (Cited on page 40.)

[84] Ronacher, A. Jinja2 (the Python template engine). http://jinja.pocoo.org/, 2011. (Cited on pages 109 and 136.)

[85] Rotem, N. C-to-Verilog. http://c-to-verilog.com/, 2009. (Cited on page 24.)

[86] Sedgewick, R. *Algorithms in Java: Part 5*, 3 ed., vol. 2. Addison-Wesley, 2003. (Cited on page 37.)

[87] Sedgewick, R., and Wayne, K. *Algorithms*, 4th ed. Addison-Wesley, 2011. (Cited on page 105.)

[88] Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., and Hanrahan, P. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph. 27*, 3 (Aug. 2008), 18:1–18:15. (Cited on page 2.)

[89] Silicon Graphics. Extraordinary acceleration of workflows with reconfigurable application-specific computing from SGI. Tech. rep., 2004. (Cited on page 11.)

[90] Singh, A., Parthasarathy, G., and Marek-Sadowska, M. Interconnect resource-aware placement for hierarchical FPGAs. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design* (Piscataway, NJ, USA, 2001), ICCAD '01, IEEE Press, pp. 132–136. (Cited on page 70.)

[91] Singh, S. New parallel programming techniques for hardware design. In *Proc. IFIP International Conference on Very Large Scale Integration VLSI - SoC 2007* (2007), pp. 163–167. (Cited on page 19.)

[92] Sinnen, O. *Task scheduling for parallel systems*. John Wiley & Sons, 2007. (Cited on page 64.)

[93] Snider, G., Shackleford, B., and Carter, R. J. Attacking the semantic gap between application programming languages and configurable hardware. In *FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable*

gate arrays (New York, NY, USA, 2001), ACM, pp. 115–124. (Cited on page 19.)

[94] Sourceforge. FpgaC compiler, 2006. (Cited on page 18.)

[95] Stevens, J. Hybridthreads compiler: Generation of application specific hardware thread cores from C. In *Proc. of the Intl. Conf. on Field Programmable Logic and Applications (FPL'07)* (2007), pp. 511–512. (Cited on page 25.)

[96] Stitt, G., and Vahid, F. Binary synthesis. *ACM Trans. Des. Autom. Electron. Syst. 12*, 3 (2007), 1–30. (Cited on page 25.)

[97] Sutter, H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal 30*, 3 (2005). (Cited on page 2.)

[98] Terasic Technologies. Altera DE4 development and education board. http://de4.terasic.com/, 2012. (Cited on page 14.)

[99] Tiwari, A., Chen, C., Chame, J., Hall, M., and Hollingsworth, J. A scalable auto-tuning framework for compiler optimization. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* (2009), pp. 1–12. (Cited on page 31.)

[100] Todman, T., Constantinides, G., Wilton, S., Mencer, O., Luk, W., and Cheung, P. Reconfigurable computing: architectures and design methods. *Computers and Digital Techniques, IEE Proceedings - 152*, 2 (mar 2005), 193 – 207. (Cited on page 12.)

[101] Tripp, J. L., Jackson, P. A., and Hutchings, B. L. Sea cucumber: A synthesizing compiler for FPGAs. In *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream* (2002), vol. 2438 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 51–72. (Cited on page 25.)

[102] Wikipedia. Handel-C. http://en.wikipedia.org/wiki/Handel-C, 2009. (Cited on page 18.)

[103] Wikipedia. Accelerated processing unit. http://en.wikipedia.org/wiki/Accelerated_processing_unit, 2013. (Cited on page 2.)

[104] Wikipedia. List of device bandwidths. http://en.wikipedia.org/wiki/List_of_device_bandwidths, 2013. (Cited on page 15.)

[105] Xenos, S. Priority queues. http://www.theturingmachine.com/algorithms/heaps.html, 2007. (Cited on page 39.)

[106] Xilinx Inc. *Vivado Design Suite User Guide: High-Level Synthesis*, December 2012. (Cited on page 24.)

[107] XtremeData Inc. *XD1000 VHDL Development User Guide*, August 2006. (Cited on page 10.)

[108] XtremeData Inc. XD1000 datasheet, 2007. (Cited on pages 10, 52, and 55.)

[109] Zhang, Z., Fan, Y., Jiang, W., Han, G., Yang, C., and Cong, J. Autopilot: A platform-based esl synthesis system. In *High-Level Synthesis*, P. Coussy and A. Morawiec, Eds. Springer Netherlands, 2008, pp. 99–112. (Cited on page 24.)