Department of Computing, Imperial College London

MEng Individual Project

Aspect Oriented Design for Dataflow Engines

June 2013

Author

Paul Grigoraş

Supervisors

Prof. Wayne Luk

Dr. Stephen Weston

Submitted in part fulfilment of the requirements for the degree of
Master of Engineering in Computing of Imperial College London

Abstract

Dataflow designs implemented on custom streaming architectures can be orders of magnitudes more efficient than traditional software, but they are developed with reduced productivity. We propose a novel design flow for generating dataflow designs based on Aspect-oriented programming and explain how the proposed approach can be used to decouple design optimisation from design specification by encapsulating optimisations in separate aspect descriptions, which leads to improved productivity and efficiency. To support this approach we introduce a novel dataflow language that facilitates integration with existing aspect weaving tools and simplifies design development by supporting embedded hardware/software co-design, flexible number representation and run-time reconfiguration. We introduce novel aspect descriptions that specify system-level and implementation level optimisations strategies as well as strategies for improving developer productivity. We evaluate our approach on a number of applications, including advanced high-performance applications, such as the Reverse Time Migration technique for seismic imaging, and show that efficient dataflow designs up to 100 times faster than equivalent software only implementations can be derived with improved productivity.

Acknowledgements

I would like to thank:

- Professor Wayne Luk, for providing the inspiration for this project and guidance throughout
- Dr. Jose Gabriel Coutinho, for invaluable feedback and suggestions and including this work in the HARNESS project and providing the original source code for the Add Predictor kernel
- Xinyu Niu, for providing the original source code and background on the Reverse Time Migration and Black Scholes applications
- Dr. Timothy Todman and Dr. Stephen Weston, for providing constructive suggestions regarding the evaluation process
- Maxeler Technologies, especially Jacob Bower and Oliver Pell for their constructive suggestions on improving our paper and approach
- the anonymous reviewers of our paper for some interesting suggestions regarding future work
- my family and friends for their continuous support

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Existing work shows that dataflow machines emulated on FPGAs achieve significant performance gains compared to multi-core processors when implementing high throughput, highly parallel applications that operate on large, uniform data sets [1, 2]. For example, an implementation of Reverse Time Migration, an advanced seismic imaging application, has been shown to be 103 times faster and 145 times more energy efficient than an optimised implementation running on a multi-core microprocessor [3]. However, the dataflow paradigm is not widely adopted and imperative languages are significantly more popular [4]. The fact that a large number of high-performance applications are written in C/C++ indicates the attraction and potential of translating programs in these languages to dataflow designs to improve performance and energy efficiency. Once the dataflow design for a particular application has been generated, it is optimised for the target architecture to maximise performance subject to constraints such as latency, power consumption or cost. Even when performed manually, this approach can result in significant performance improvements. Therefore, enhancing productivity of developers using this approach would have a significant impact and is exactly what this project would address.

1.1 Motivation

The translation and optimisation process is laborious and involves many manual, slow and error-prone steps, that can include even a complete redesigning of the original algorithm. Additionally, high-performance applications rely heavily on complex architecture specific optimisations. Transformations that enable hardware/software partitioning [5], low level optimisations (e.g. operator bit width [6]) or high level optimisations (such as loop unrolling [7], blocking and tilling [8]) obscure the application's original purpose, reducing maintainability. Furthermore, since some optimisations depend on platform specific properties (e.g. bit width of Digital Signal Processors), mixing them with the

application code affects portability, complicating the process of targeting different platforms without repeating the optimisation process. Finally, due to the large number of design choices, an incomplete manual exploration can lead to sub-optimal results. Considering these issues in the context of cloud computing solutions that are expected to provide heterogeneous acceleration for a plethora of customer applications, the need for fully automated and portable code generation for high-performance designs becomes apparent. In addition, the ability to automatically explore various trade-offs between price, performance and energy efficiency would provide cloud owners with flexibility in implementing their business offering.

Hence it is important to decouple these transformations from the application logic, allowing them to be reused on multiple platforms and facilitating design space exploration. Using Aspect Oriented Programming [9], this can be achieved by separating cross-cutting concerns, such as optimisations and structural transformations, from the main algorithm and encapsulating them in separate aspect descriptions. Additionally, aspect descriptions can be used to capture non-functional requirements which may include constraints on power consumption, data rate, latency, execution time etc. For example in a cloud computing environment which manages many applications simultaneously, automated optimisation techniques based on aspect descriptions can be used to adapt implementations to fit long or short term power and performance goals. These strategies could exploit the run-time reconfiguration potential of FPGA chips to vary between existing implementations based on input characteristics, or map the applications to different accelerators.

Hence, the importance of the project lies in the potential to:

1. **improve performance of existing imperative applications by orders of magnitude**, by automating the translation to FPGA dataflow designs and providing an automated aspect-driven framework for exploring the design space of optimisation techniques applicable to fully customisable acceleration devices;

2. **address portability issues of massively parallel applications**, by using a novel aspect-driven synthesis flow which allows specification of platform independent optimisation strategies, possibly addressing global challenges in Exascale computing by allowing a portable expression of parallelism and data locality [10, pp. 46 – 50];

3. **greatly increase developer productivity**, by automating the translation and optimisation process thus removing the potential for human error and facilitating the reuse of portable designs and optimisation strategies which could significantly reduce development time, given the lengthy compilation process on high-end FPGA devices [11, pp. 23 – 28].

Consequently, the project could significantly improve performance of state of the art dataflow designs and simplify the development and maintenance of applications in key areas where high-performance dataflow computing is used such as finance [12, 13], geo-

science [14], weather forecasting [15] and cloud computing [16].

1.2 Challenges

Combining the dataflow paradigm with an aspect driven synthesis flow could be expected to result in improved performance, portability and developer productivity. We believe that the following challenges need to be addressed to facilitate the adoption of dataflow designs:

1. **Specifying dataflow designs**, in an intuitive, well understood language that is concise, facilitates the translation of existing designs, and is sufficiently expressive to support the requirements of modern high-performance applications. Compared to the solution used in [17], this requires significant structural transformations to be captured and applied via aspect descriptions to transform the original imperative source into a dataflow implementation. However, results show that significant speedups can be achieved by targeting streaming architectures [18].

2. **Specifying optimisation strategies**, decoupled from the application code in a manner that makes the specification easy to reuse and to customise, and is comprehensive enough to allow capturing of optimisations at various levels: *algorithmic transformations* of the original application that expose parallelism or improve communication between CPU and accelerator, *design-level transformations* that enable exploration of platform specific optimisations and *productivity related transformations* that improve developer productivity. Existing implementations for streaming architectures often require intricate specialisation steps to adapt and optimise the original application. These can be encapsulated in aspect descriptions and reused for multiple applications, simplifying the design process. For example, [3] shows that using run-time reconfiguration to swap an application's idle sections with useful functions at run-time can lead to a 45% performance improvement. Aspect-driven synthesis could be used to automate this optimisation, enabling its application for a broader class of programs.

3. **Systematic design space exploration** of dataflow designs driven by these parameterizable optimisation strategies, that increase developer productivity and allow exploration of design level trade-offs.

4. Applying these design techniques to **create and optimise high-performance applications**.

In this project we investigate how these challenges can be addressed by using an aspect-driven synthesis and optimisation process to improve both design performance and designer productivity. This is achieved by allowing the specification of reusable optimisation strategies, and decoupling them and platform specific transformations from the original application making it easier to maintain and more portable. We study how tech-

niques of Aspect Oriented Programming can be applied to the compilation and optimisation of designs targeting streaming Data Flow Engines (special computing devices built around Field Programmable Gate Array chips that implement the dataflow paradigm of computation). This includes identifying efficient compiled patterns for streaming DFEs and how such patterns can be obtained from high level descriptions using AOP design techniques.

We identify the following objectives:

1. *Introduce a design flow that can be applied to integrate dataflow tools for FPGAs with flexible aspect weaving and design space exploration tools*
2. *Identify novel aspect descriptions that enable design space exploration of dataflow designs from high level descriptions.*
3. *Capture dataflow and platform specific optimisations using aspect descriptions and apply them to the generated designs.*
4. *Evaluate the approach by implementing advanced high-performance applications.*

In the next section we identify how these objectives can be met.

1.3 Contributions

We propose a methodology for addressing the challenges and meeting our objectives based on the following contributions:

1. We propose a novel, automated method for design space exploration of dataflow designs, driven by optimisation and transformation strategies captured with aspect descriptions. This design flow is introduced in chapter 3.
2. We introduce FAST, a novel dataflow language based on C99 syntax that can be used to create high-performance designs. The language is used as part of the proposed design flow to implement dataflow kernels. Chapter 4 provides an overview of the FAST language.
3. We present novel aspects for specifying optimisation strategies at the system level, the implementation level, the exploration level and development level. We implement these aspects using LARA, an aspect oriented language for embedded reconfigurable systems. We present the aspect descriptions in chapter 5.
4. We implement a compiler that translates designs in FAST to MaxCompiler designs which are then compiled and executed on a Maxeler MaxWorkstation containing a MAX3 DFE with a Virtex 6 FPGA chip and 24GB of DRAM.. We present an overview of our implementation in Chapter 4.5.

5. We evaluate our approach by implementing a high-performance design for an application based on the Reverse Time Migration technique for seismic imaging [3]. We discuss the results in Chapter 6.

1.4 Published Work

As part of the project a full paper has been accepted for publication at the 24th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP 2013[1]. The paper "Aspect Driven Compilation for Dataflow Designs" [19] is based on material from Chapters 3, 4 and 5 of this report and introduces the proposed design flow, the FAST language and a number of aspect descriptions for improving productivity and analysing resource trade-offs.

A second full paper is being drafted for submission at the 2013 International Conference on Field-Programmable Technology, ICFPT 2013[2]. This is based on material from Chapter 5 and introduces aspect descriptions for run-time reconfiguration.

Finally, the approach and software implementation described in this report were included into the FP7[3] funded HARNESS Project. HARNESS[4] aims to integrate heterogeneous computing resources into cloud platforms in order to reduce energy consumption and increase performance and cost effectiveness of key cloud application in areas such as finance and geoscience. The FAST language and `fastc` compiler will be used in the process of generating efficient dataflow implementations for key algorithms based on cloud owner requirements.

[1] `http://asap-conference.org/`
[2] `http://www.fpt2013.org/`
[3] European Union Seventh Framework Programme
[4] `http://www.harness-project.eu/?page_id=21`

Chapter 2

Background

In this chapter we compare streaming data flow architectures with traditional, general purpose architectures and we describe the Maxeler hardware acceleration solution and the MaxCompiler toolchain and API which represent the target of the MaxC compilation process. We also look at the LARA language which will be used as part of the design flow to specify and apply optimization strategies both to the original source code and to the resulting dataflow design. We present related work in the areas of high level synthesis tools, dataflow languages and aspect-driven compilation for FPGA designs and explain how our approach differs from existing work in these fields.

2.1 Dataflow Computing

Although general purpose computing devices offer a convenient programming paradigm, the traditional fetch - decode - execute cycle is inherently sequential and relies on inefficient access to external memory. To compensate for this a large area of a modern CPU core is dedicated to caches, branch prediction units and out-of-order scheduling and retirement units. This reduces the area of the chip available for performing useful computation. Furthermore, although multicore programing is an answer for the processor power wall (which prevents increases in operating frequency beyond a certain point, limiting the processing speed of a single core device), there are classes of algorithms whose performance does not scale linearly with the number of cores. This is especially true when operating on large volumes of data with poor spatial locality that do not fit into a CPU's on-chip cache such as algorithms involving sparse matrix computations or convolution [20]. Although this model offers good flexibility when dealing with arbitrary access patterns, it is not efficient for large volumes of highly regular data.

The dataflow computing paradigm operates differently form the general purpose computing paradigm (as shown in Figure 2.1), being designed to be efficient at processing

large volumes of data. It works by creating a streaming dataflow graph of computational nodes, which operates as a large computational pipeline: input data is streamed in sequentially through each pipeline stage and output data is streamed out. This results in a highly pipelined design that can be statically scheduled achieving throughput rates of one value per cycle by completely avoiding pipeline hazards. This means that a design running at a few hundred megahertz can outperform a CPU implementation running at a few gigahertz while being more energy efficient [18].

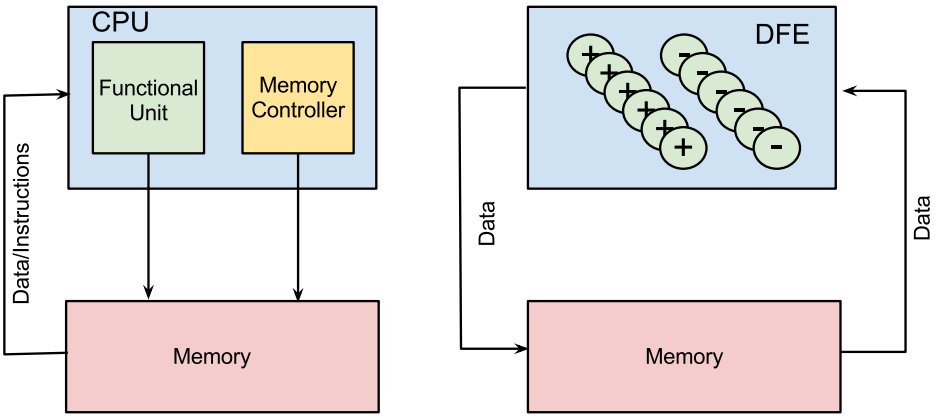

Figure 2.1: Comparison between general purpose CPU architecture and a streaming Data Flow Engine. In the case of the latter instructions are not stored in memory but encoded in the dataflow graph.

2.2 FPGA Acceleration

Dataflow computing is a general computing paradigm that could be applied to various architectures such as multi-core, GPUs, Cell processor arrays or ASICs. In this project we chose to focus on FPGAs for the following reasons:

- FPGAs provide greater **flexibility** and lower time-to-market than ASICs;
- FPGAs present exciting **low-level optimisation opportunities** such as word-length optimisations or exploiting run-time reconfiguration potential to improve performance at run-time;
- FPGAs are considerably **more power efficient** than multi-core CPUs and GPUs;
- FPGAs offer a **predictable performance model**, due to static scheduling and

lack of thread barriers or other traditional synchronisation mechanisms;

One major limitation of FPGAs compared to CPUs and GPUs is the long development time. This is caused by:

- **large compilation time** – compilation time of a single FPGA design can vary from 20 minutes to several days even on multi-core clusters with large amount of memory can range from);
- **lack of high-level tools** – although high-level tools such as MaxCompiler (Section 2.2.3 exist that can automate the build process and simplify considerably the design of dataflow pipelines, these are not high level enough to permit expressing algorithms in an intuitive, well-understood approach; additionally they do not abstract the underlying hardware architecture, the programmer having to constantly deal with adjusting the latency of various design components, perform optimisations that allow the design to build at higher frequencies etc.;
- **large design space** – this leads to the need to explore a large design space which is a time consuming process, especially given long compilation times;

2.2.1 Architecture

FPGAs are logic chips that can be reconfigured in seconds to implement custom applications. Hence they offer a much shorter time to market than traditional ASIC[1] based solutions, while still being able to implement custom logic circuits, making them significantly faster than general purpose hardware. However, the size of the FPGA chip constrains the design that can be uploaded onto the chip. FPGAs have a limited number of each of the following resource types:

- **look-up tables** (LUTs) - implement the logical functions performed by the circuit;
- **flip-flops** (FFs) - small storage elements;
- **digital signal processors** (DSP) - small special purpose arithmetic units;
- **block RAM** (BRAM) - larger, on-chip storage elements.

2.2.2 Run-time Reconfiguration

Run-time reconfiguration refers to the capability of uploading a new design onto the FPGA at run-time. This capability can be used to maximise usage of the FPGA accelerator for two purposes:

1. **enabling** – to enable the execution of functions that would normally not fit on

[1] Application Specific Integrated Circuit

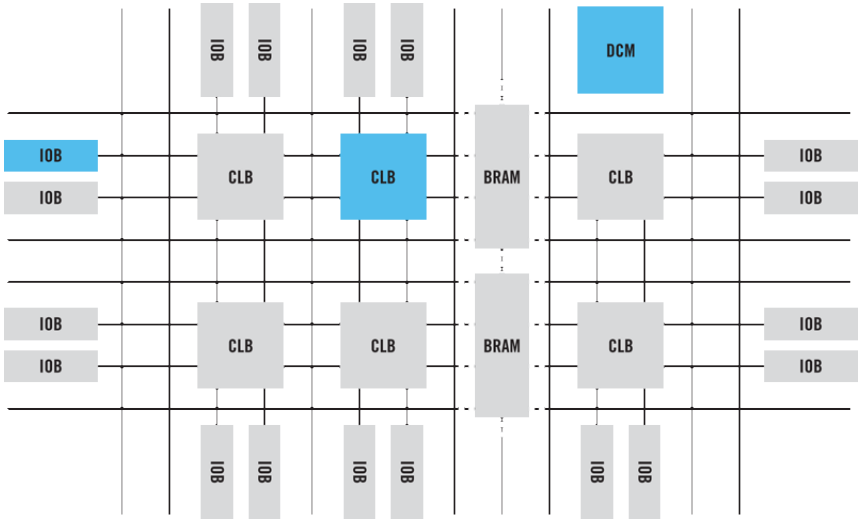

Figure 2.2: Structure of an FPGA block: Configurable Logic Blocks (CLB), Interconnect, Input/Output Blocks (IOB) and Embedded Memory (BRAM). Source: `http://www.origin.xilinx.com/fpga/`.

the accelerator; this can be achieved for example via time-sharing, where a design is partition into smaller tasks that run sequentially on the accelerator[21]

2. **optimisation** – to improve performance of FPGA designs, by removing idle functions; this can be achieved by removing, at run-time, idle portions of a design which allows one to replicate the useful functions in order to increase overall performance

For the second type of application we consider the example of a an FPGA sorting architecture presented in [22]. A parallel sorting tree is used for small problem sizes that fit the FPGA on-chip memory (a few hundred inputs). While this network achieves a very high throughput, it does not scale with the number of inputs, and for larger input sizes it exceeds the available FPGA resources. Hence, a solution is to combine this network with a FIFO-based merge sorter[23] which allows the merging of sorted buckets of numbers. This leads to a two step sorting algorithm, using run-time reconfiguration, in which during the first stage a small number of inputs are merged in parallel and in the second stage, resulting buckets of inputs are merged to obtain the final sorted sequence.

An example of the first type of application is shown in [3] run-time reconfiguration is used to maximise performance by removing idle design components. This works well in multi-step algorithms such as the Reverse Time Migration application which has 2 major steps: a forward and a backward propagation of a simulated wave through the earth surface. Thus the functions that would only be used in the second stage need not be uploaded on the accelerator from the very beginning. This enables the an increase in resources used for the first part which can be used to increase parallelism and thus reduce computation time.

Based on the reconfiguration technique, we distinguish between

- **total reconfiguration**, where the entire FPGA is reconfigured with a new design; this exhibits a high reconfiguration time often in the are of seconds and requires that data be saved from on-board DRAM to host memory during the reconfiguration process, which adds significant overhead;
- **partial reconfiguration**, where a smaller region of the FPGA is reconfigured, reducing reconfiguration overhead, eliminating the need for buffering data and minimising computation stall

The Maxeler Platform used in this projects was not specifically designed for run-time reconfiguration. As such it relies on a large FPGA chip for which the reconfiguration overhead is approximately 0.7 - 1 second, depending on the size of the design. In addition, the slow data transfer rate and large transfer latency over PCI-Express from the accelerator card to the memory of the host system introduces complicates the process of improving performance via run-time reconfiguration.

The Maxeler Platform does not currently support partial run-time reconfiguration, so the possibility of using partial run-time reconfiguration to improve design performance is left as future work. However, given that partial reconfiguration overhead is smaller than that of total reconfiguration, our approach will operated similarly or even better in the case where partial reconfiguration capabilities are available.

2.2.3 Maxeler Platform

Maxeler Technologies provides a software and hardware acceleration solution based on the dataflow computing model. The dataflow design is created using MaxCompiler [24] and implemented on a specialized hardware platform, built around high-end Field Programmable Gate Array (FPGA) chips.

The specific data flow engine used for this project is a MAX3424A card based on a Virtex 6 FPGA chip [25]. The MAX3 provides 24GB of on-board DRAM and about 4MB of fast on-chip BRAM are available on the FPGA chip.

The system is connected to the dataflow engine via PCIe as shown in Figure 2.3.

Since the target of our compilation process is a MaxJ/MaxCompiler design, we provide a brief summary of the most important features in the rest of this section.

We demonstrate the use of MaxCompiler in accelerating a simple moving average computation, starting from an original design in C shown in Listing 2.1. This performs a three point moving average computation on an input array x, using 2 point averages at boundaries.

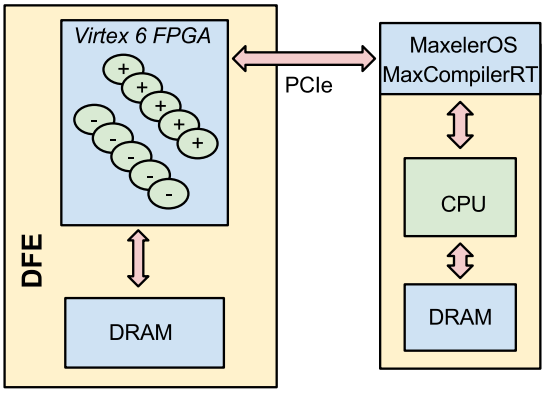

Figure 2.3: The Maxeler acceleration solution: the DFE is connected to the host machine via PCIe. The board comprises 48GB of DRAM and a Virtex 6 FPGA chip.

```
1  for (int i = 0; i < n; i++ ) {
2    sum = x[i], divisor = 1;
3    if ( i > 0 )
4    sum += x[i - 1], divisor++;
5    if (i < n - 1)
6    sum += x[i + 1], divisor++;
7    y[i] = sum / divisor;
8  }
```

Listing 2.1: Original three point moving average computation in C.

MaxJ

MaxJ is a high-level language for specifying dataflow architectures. It is largely based on Java and adopts a meta-programming approach for specifying dataflow graphs. Listing 2.2 shows the dataflow kernel for computing a three point moving average over an input stream, which highlights some of the important features of the MaxJ language:

- kernel inputs and outputs provide an I/O interface that allow the kernel to communicate with the rest of the design (Lines 4 and 12);
- stream offset expressions allow accessing past and future elements of a stream (Lines 7 and 8). The offset window is stored into on chip BRAM so is limited to a few tens of thousands elements;

- frequently used components such as counters are provided by the API. They are useful in keeping track of iteration count when mapping loops to streaming designs (Line 5);
- operator overloading is used to perform arithmetic on input streams;
- multiplexers (in this instance represented by the overloaded conditional operator) are used to select between streams (Lines 11 and 12).

```
1  public class MovingAverageKernel extends Kernel{
2    public MovingAverageKernel(...) {
3      super(...);
4      HWVar x = io.input("x", hwFloat(8, 24));
5      HWVar cnt = control.count.simpleCounter(32, N);
6
7      HWVar prev = cnt > 0       ? stream.offset(x, -1) : 0;
8      HWVar next = cnt < (N - 1) ? stream.offset(x, +1) : 0;
9      HWVar divisor = cnt > 0 & cnt < (N - 1) ? 3.0 : 2.0;
10
11     HWVar y = (prev + x + next) / divisor;
12     io.output("y" , y, hwFloat(8, 24) );
13   }
14 }
```

Listing 2.2: Kernel design for the three point moving average computation showing features such as stream offsets, arithmetic and control which will have to be supported by our MaxC language

MaxCompiler

MaxCompiler [26] is a high level compiler targeting the acceleration platform developed by Maxeler Technologies. It provides a Java based API for specifying hardware designs that are compiled and uploaded onto the DFE and a C runtime interface (MaxCompilerRT and MaxelerOS shown in Figure 2.3) for the part of the application running on the CPU of the host system.

In addition to the dataflow kernels, a MaxCompiler design also contains a manager design which is sude manage kernel I/O, connecting multiple kernels together (in multi kernel designs) and kernels to DRAM and the CPU interface (via PCIe).

A number of dataflow kernels are connected via a manager to create a dataflow design which is then compiled to the a Xilinx bitstream that can be uploaded to the FPGA. Additionally a number of run-time functions are generated that can be called by the MaxCompiler run-time to load the design and initiate the streaming of data to and from the dataflow design.

An example manager design is shown Listing 2.3 and is used to instantiate a single moving average kernel and connect its inputs and outputs to the host interface.

```
1  public class MAManager extends CustomManager {
2    public MovingAverageManager(...) {
3      super(...);
4      KernelBlock k = addKernel(new MovingAverageKernel(...));
5      k.getInput("x") <== addStreamFromHost("x");
6      addStreamToHost("y") <== k.getOutput("y");
7    }
8  }
```

Listing 2.3: Manager design for the three point moving average computation

Run-time

Finally we must write a host application which is required to queue the input streams and run the accelerator. This is achieved by calls to the MaxCompilerRT (runtime) API that interfaces with MaxelerOS.

```
1  max_run(
2  device,
3  max_input("x", x, x_size),
4  max_output("y", y, y_size),
5  max_runfor("MAKernel", n),
6  max_end());
```

Listing 2.4: Host example for queueing the input and output streams and running the three point moving average design.

Analysis

Although the MaxCompiler toolchain greatly simplifies the process of accelerating applications and particularly the designing of dataflow kernels, the acceleration process is still very involved and requires a large amount of experience with FPGA technology and domain specific knowledge. Most importantly the whole process is manually driven including the exploration of optimizations. This step is a critical and time consuming part of the design process which is vital in achieving maximum performance (in terms of operating frequency, number of parallel pipelines etc.) subject to physical limitations such as chip size or timing constraints.

By specifying the design in FAST and optimization strategies in LARA we aim to create

the basis of a design space exploration flow that can automate this process.

2.3 Stencil Computation

Stencil computation is a class of computational problems consisting of iterative kernels that update array elements based on the values of their neighbours. The pattern of operation is called a stencil.

Stencil computation has important applications in solving Partial Differential Equations and computer simulations (such as heat diffusion).

They can operate on multiple dimensions. But increasing the dimension heavily impacts memory reference locality, so higher order stencils are hard to compute on CPU and have been the focus of many optimisations.

We present some background on a number of applications that can be reduced to 1, 2 or 3D stencil computation that are used as part of our benchmark suite (Chapter 6). We illustrate the importance of the applications and traditional approaches to accelerating them.

2.3.1 Numerical Differentiation

For an example of a 1D stencil we consider the problem of numerical differentiation [27], which provides a method for estimating the derivative of a function based on its values at discrete points. By replacing the derivative with finite difference expressions the following 5 point linear stencil can be derived to approximate the value of $f'(x)$:

$$f'(x) = \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h} \tag{2.1}$$

Such an approximation is useful for computing the derivative of a function based on a discrete set of observed values (e.g. to compute speed/acceleration of an object based on a discrete sample of displacements/speeds). It provides better accuracy at the cost of double computational cost when compared to Newton's difference quotient $f'(x) = \frac{f(x+h)-f(x)}{h}$ or the slope of secant method $f'(x) = \frac{f(x+h)-f(x-h)}{2h}$.

Hence the following stencil algorithm could be used to compute the numerical approximation of the derivative of f for a large number of points in the function's domain:

Algorithm 1 Stencil approximation of first order derivative

function NUMERICALDIFFERENTIATION($f, Order$)
  $h \leftarrow \epsilon$
  for $x \in Order, nPoints - Order$ **do**
    $f'(x) \leftarrow \frac{-f(x+2h)+8f(x+h)-8f(x-h)+f(x-2h)}{12h}$
  end for
  return f'
end function

2.3.2 Black Scholes Equation

Stencil computation can also be used for solving Partial Differential Equations using finite difference methods. This has many applications in scientific computing for finance and physics for example, where PDEs are used to model the dynamic behaviour of a system. For example, the solution to the widely-used Black Scholes equation can be approximated using a 2D stencil computation.

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0 \tag{2.2}$$

where:

- S - stock price,
- V(t, S) - price of derivative as a function of stock price and time
- r - risk-free interest rate
- σ - volatility of the stock's returns

A finite difference approximation using the explicit method leads to a recursive formula [28] for $v(i, j)$, the approximated value of V at the ith point in the time dimension and the jth point in the stock price dimension:

$$v(i, j) = \alpha * v(i, j - 1) + \beta * v(i, j) + \gamma * v(i, j + 1) \tag{2.3}$$

This leads to a 2D stencil algorithm that performs a forward propagation on the time parameter by iteratively computing the values of $v(i + 1, j)$ for all j, given the values of $v(i, j)$ for all j as show in Algorithm 2.

Although the shape of the stencil remains one dimensional, by contrast to the numerical differentiation example, the computation is also propagated forward in time, adding another (time) dimension. Thus this can be viewed as a 2D computation.

Edge detection algorithms can be used to detect points at which sudden changes in image

Algorithm 2 Stencil kernel for finite difference approx. of Black Scholes PDE

```
function BlackScholes(payoff, timesteps)
    for i ∈ 0, timesteps do
        for j ∈ 1, size(payoff) - 1 do
            payoff_next[j] ← α∗ payoff[j−1] +β∗ payoff[j] +γ∗ payoff[j+1]
        end for
        payoff ← payoff_next
    end for
    return payoff
end function
```

brightness occur. These images

2.3.3 Reverse Time Migration

Stencil computation is also used for advanced simulation and imaging algorithms. For example the Reverse Time Migration technique is used to detect geological structures beneath the Earth's surface [29] and is widely used in the Oil and Gas sector. The algorithm uses a forward propagation step which models the propagation of injected acoustic waves with the isotropic acoustic wave equation [30]:

$$\frac{d^2p(r,t)}{dt^2} + c(r)^2 \bigtriangledown^2 p(r,t) = f(r,t) \tag{2.4}$$

where:

- c(r) - sound speed model
- p(r, t) - seismic wave value at point r, time t
- f(r, t) - input wave value at point r, time t

An approximation using finite difference is given in [31] and leads to the 3D stencil kernel [3] shown in Algorithm 3.

2.4 Aspect Oriented Programming

Aspect-Oriented Programming [32] is a programming paradigm which is used to capture aspects that cut through levels of abstractions. A commonly used example is that of program logging [33], which represents a concern that is expressed at, for example, every method entry and exit point. The functions that perform logging are spread throughout the code and, for example, ensuring consistency when changing the logging format can be difficult. Encapsulating crosscutting concerns in aspects can improve cohesion, making

Algorithm 3 Stencil Kernel for Reverse Time Migration.

```
for t = 0 ← nt-1 do
    for x = 0 ← nx-1 do
        for y = 0 ← ny-1 do
            for z = 0 ← nz-1 do
                p(t,x,y,z) = c *(
                c0 * p(t,x,y,z) +
                c11* (p(t,x-1,y,z) + p(t,x+1,y,z)) + ... + c15*(p(t,x-5,y,z) + p(t,x+5,y,z)) +
                c21* (p(t,x,y-1,z) + p(t,x,y+1,z)) + ... + c25*(p(t,x,y-5,z) + p(t,x,y+5,z)) +
                c31* (p(t,x,y,z-1) + p(t,x,y,z+1)) + ... + c35*(p(t,x,y,z-5) + p(t,x,y,z+5)) +
                d0 * p(t,x,y,z) + d1 * p(t-1,x,y,z-1) + f(t,x,y,z);
            end for
        end for
    end for
end for
```

the code easier to understand and change which leads to improved maintainability and increased developer productivity. The model which governs the application of aspects to code is named *join-point model* and describes where in the application (the *pointcut*) should the functionality implemented in the aspect (the *advice*) be applied.

This idea can also be applied to the compilation of dataflow designs, by regarding the optimisations and transformations that are applied to the original design as cross-cutting concerns. These optimisations can often hide the original purpose of the application, making it harder to maintain and by encapsulating them in aspects we not only improve the maintainability of the application but also simplify the process of optimisation and take important steps towards fully automating the optimisation and transformation process.

However, most commonly used implementations of Aspect-Oriented Programming, such as AspectJ [34, 35] and AspectC++[36] share a limitation which makes them impossible to use for capturing design optimisations and transformations, without substantial extensions: aspects cannot be applied to multiple statement blocks, loops or conditionals.

Hence, for the purpose of this project we use LARA, an Aspect-Oriented Programming language that was introduced in the scope of the REFLECT[37] project, also to support the process of optimising FPGA designs.

2.4.1 LARA

Lara is an Aspect-Oriented Programming language for specifying compiler strategies for FPGA-based systems [17]. LARA facilitates decoupling of optimisations from apllication code and enables the capturing of strategies for:

- *instrumentation*, *monitoring* and *hardware/software partitioning*, used in the initial stages of our design flow to identify optimisation candidates for mapping onto the

dataflow engine

- code *specialisation* and *optimisation*, used both for the original host application and dataflow designs which are the input of our design flow

Furthermore LARA descriptions can be parametrised to simplify integration with our proposed design space exploration step described in Chapter 3.

Listing 2.5 shows an example description used for fully unrolling all innermost loops with an iteration count smaller than or equal to 16. The 'select' statement on line 2 captures the join points on which the aspect acts, the 'apply' statement specifies actions to be applied to the results of a query while 'condition' is used to filter relevant queries.

```
1  aspectdef loopunroll
2  select function.loop{type="for"} end
3  apply optimize("loopunroll", "fully"); end
4  condition $loop.is_innermost && $loop.num_iter<=16; end
5  end
```

Listing 2.5: Example LARA description for fully unrolling innermost loops with an iteration count smaller than or equal to 16.

[38] also introduces a design flow based on LARA for mapping applications into heterogeneous multi-core platforms. This involves specifying optimisations and mapping strategies in the LARA programming language, separate from the application source code.

The strategies are compiled and applied to the original source code in a sequential order to obtain the final design which is synthesised using Catapult-C [39]. Examples of strategies which are expressed using LARA include loop unrolling, coalescing, loop fission and mapping compute intensive functions to hardware (hardware/software partitioning).

Advantages of the aspect based approach compared with the popular pragma based approach used in frameworks like OpenMP [40] include the possibility of defining dynamic join points through which strategies can be applied to the intermediate results of the weaving (source translation) process. Additionally, optimisation strategies are grouped into cohesive aspects, rather than being spread through the code which makes them easier to understand, maintain and modify, for example to target a different platform.

2.5 Related Work

2.5.1 High Level Synthesis

Substantial work has been carried out in synthesising high level languages to hardware designs and many tools exist for this purpose [41, 42, 43]. However these approaches do not target a streaming dataflow architecture but either soft processor designs - processor cores implemented on the FPGA chip with configurable custom computing units (e.g. floating point units). These usually offer limited speedups when compared to high-end hardcore CPUs but can turn out to be more energy efficient.

[44] proposes a method for synthesising hardware pipelines from OpenCL programs which exploits some important concepts related to parallelism exposed by the OpenCL specification [45] such as threads and domain decomposition into threads sharing local memory. Although we are dealing with simple C kernels for this project, some of the proposed compilation strategies can be applied for loops.

2.5.2 Dataflow Languages

A number of dataflow languages have been developed targeting FPGAs but also multi-core platforms.

Lucid [46] (implemented in [47]), SISAL [48, 49], and Lustre [50], are examples of functional dataflow languages. The latter is based on a synchronous programming model, facilitating safety verification for critical software [51] rather than performance. The functional programming style complicates the translation of existing imperative applications and none have existing implementations for FPGAs, so a performance comparison is not possible. Control flow extensions to dataflow languages have also been investigated in [52] and [53]. This shows that in most real life applications, it is necessary to specify a control path as well as a data path.

Streams-C [54] and ImpulseC [43] adopt imperative ANSI C syntax and an execution model based on Communicating Sequential Processes and introduce non-standard syntax and constructs for specifying designs such as special comment blocks which are used to annotate the C application code. The specialised syntax makes the languages harder to integrate with existing source-to-source translation or aspect weaving frameworks.

Hybrid approaches such as MaxCompiler [26] separate the CPU run-time component from the accelerated one, providing a C run-time environment and a Java API for building dataflow designs via meta-programming. The separation complicates the development process, hindering sharing of design parameters and, consequently, the design space exploration process. The use of meta-programming simplifies design parametrisation, but can make resulting programs harder to understand. In contrast, the pro-

posed approach allows the computation description, which includes CPU and dataflow components, to be specified using a single language and to be decoupled from design parametrisation and other optimisation strategies which are captured as LARA aspects. This separation of concerns results in more intuitive and maintainable descriptions.

2.5.3 Aspect-driven Compilation of FPGA Designs

One attempt at capturing optimisation strategies for FPGA designs using aspect descriptions is LARA [55, 38], an Aspect Oriented language for embedded systems. Aspect descriptions are written in the LARA language and automatically applied to the original application source to generate optimised versions through various transformations that enable and optimise hardware/software partitioning [5]. Aspect descriptions can be used to specify compilation strategies that result in overall speedups of 2 to 6.8 times over software versions [17], generally with high aspect bloat [2] [55].

The use of LARA aspects in guiding the compilation process of C applications is described in [17] and [56] but the backend compilation targets a von Neumann architecture (with a GPP and custom accelerator) unlike the dataflow architecture proposed in this project. The approach described in [17] and [56] relies more on high-level source transformation whereas our approach is based on a systematic design space exploration process, which enables the analysis of more low-level optimisations. Finally, [17] and [56] do not consider development aspects which can be used to improve developer productivity.

The use of aspect-oriented programming for specifying strategies for run-time adaptation of FPGA designs discussed in [57] differs from the static process considered in this paper in which the application is partitioned and scheduled at compile time, to achieve optimised performance as described in [3]. An advantage of our approach is that an optimised allocation is generated prior to application execution. However, we lack the flexibility of adapting the design to varying input conditions.

2.6 Summary

We have provided a brief introduction into the area of dataflow computing and explained why dataflow acceleration can improve performance and energy efficiency over traditional software only solutions. We have shown the main benefits and drawbacks of FPGA acceleration and explained why we have chosen to target custom streaming architectures with our design flow. We have introduced the Maxeler platform which will be used as the backend of the compilation process presented in Chapter 3 and presented some of the more important features of the MaxCompiler approach including some limitations that we attempt to address with the proposed flow in order improve developer productivity

[2] aspect bloat $= \frac{\text{size of transformed code}}{\text{size of original code}}$, so a higher aspect bloat is better

and possibly reveal new opportunities for design transformation and optimisations. As an example of parallel computation where FPGAs can be used to obtain substantial speedups we have presented stencil computation. Finally, we briefly introduced related work in the areas of dataflow languages and aspect-driven compilation for FPGA designs and shown that our approach differs from existing work by:

- introducing a novel approach for generating and optimising dataflow designs based on an aspect-oriented compilation flow
- focusing on separating optimisations from functional application code with the view of improving both maintainability and performance
- focusing on a streaming, high-throughput architecture based on FPGA Dataflow Engines
- introducing explicit support for specifying run-time reconfiguration strategies to improve performance and energy efficiency

Chapter 3

Design Flow

In this chapter we introduce a novel design flow for creating high-performance dataflow designs starting from C/C++ applications. We explain the motivation and requirements for the proposed approach and provide an overview of its three main components:

- the *FAST language*, used to express dataflow kernels
- the *aspect description repository* and *weaver*, which group and apply the optimisation and transformation strategies encapsulate through aspects
- the *compilation backend*, used to generate FPGA bitstreams from dataflow designs and link the host code run-time application

We show how these three components can be integrated to produce an automated design technique. We analyse the steps required to produce an optimised design using the proposed approach and compare this with alternative approaches using other start-of-the-art technologies. Finally, we present an extension to our original design-flow to support run-time reconfiguration.

3.1 Design Goals

Our design flow aims to improve both *efficiency* (in terms of performance and energy consumption) and *productivity*. The former is crucial to High Performance Computing, the latter helps reduce development cost and time and is a well-known issue with existing FPGA based acceleration solutions [58]. To achieve this we focus on maintaining or improving the *performance* and *energy efficiency* of existing applications while using a more systematic approach for design optimisation that results in more *portable* application code, improves *integration* with existing applications and *automate* time consuming and error-prone tasks that need to be performed manually using traditional tools.

3.1.1 Performance and Energy Efficiency

When targeting HPC applications it is crucial that our design flow results in high-performance, highly efficient designs. By analysing existing implementations for advanced high-performance applications such as Reverse Time Migration (Appendix B), we identify key requirements of these designs:

- **Computation is the most significant part** of high-performance applications. For stencil computations this is usually a fairly large stencil operation over a large number of adjacent data-points (multiple points in each dimension). For example, the RTM kernel uses two stencil operations of 31 floating point additions, 18 floating point multiplications and 1 subtraction (Appendix B, Lines 87 and 128). Additionally, these are replicated `Par` times (with Par as large as 12) which leads to a total of $((31+18+1)*2*12) = 1200$ floating point operations to be executed on each kernel cycle (in reality floating point operations are pipelined across 13 kernel cycles, but this is, generally, transparent to the user). Hence, our approach must provide a clear and concise manner to express computation (preferably standard operators).

- Another crucial aspect in achieving high-performance is **replicating the design** across the chip to maximise speedup subject to maximum available bandwidth. This makes use of FPGA cache (around 4MB of fast-local memory) to reuse data across time steps. This pipeline replication is achieved through design parametrisation and loops whose bounds are known at compile-time. The parameters that indicate the replication factor are shared across the compute and memory kernels but also across the CPU code.

- High-performance designs **maximise usage of on-board DRAM**. Especially for stencil type computations where a number of time step iterations are performed over the original data, it is crucial to have the data available in the large, high bandwidth on-board DRAM rather than on the CPU side. Bandwidth of the on-board DRAM is 40GB/s compared to 2GB/s over PCI-Express to CPU. However, supporting DRAM introduces the additional complexity of managing multiple kernels (since memory read and write commands are, preferably, generated from kernels separated from the original dataflow kernel – see Appendix C) and. Additionally, special API calls for generating the read/write commands have to be supported (Appendix C, Line 19).

- **Expose optimisation opportunities for backend tools**. For example, our MaxCompiler backend supports some degree of trade-offs when mapping computation to DSPs. Additionally various trade-offs can be achieved by disabling pipeline depth etc. These optimisations present interesting opportunities for trade-offs that enable developers to increase the number of parallel pipelines on the chip by balancing resource usage, or possibly reducing clock frequency.

- Additionally, recent work shows the interesting possibility of improving design performance and energy efficiency for RTM by using run-time reconfiguration to remove idle functions has been recently showed that run-time reconfiguration can be used to improve the performance and energy-efficiency of FPGA designs. It is therefore important to facilitate the specification of strategies and designs that support run-time reconfiguration to maximise performance, both in the static flavour presented in [3] where partitions are generated and scheduled optimally at compile-time but also in a dynamic, self-adaptive fashion [57] in which the application can dynamically adapt itself by run-time reconfiguration based on input values.

3.1.2 Productivity

Portability

In the context of our design flow *portability* refers to two different aspects:

- *portability of transformations and optimisations* – it should be possible to reuse optimisation strategies in the context of different applications with as less user input as possible; in the context of RTM this is no longer possible: the transformation steps from the original naive implementation have been lost, and are now part of the existing code base. Of course these could be recovered, from version control for example, but a version control patch could obviously not be applied to other applications to generate highly pipelined designs. In other words if the developer was faced with the task of accelerating a similar computational kernel (which is likely since stencil computations follow a clear pattern) he would have to re-do all steps, resulting in a large decrease of productivity.
- *portability of designs over various FPGA devices* – generally resource available to FPGA chips vary greatly based on the chip manufacturer. Even in the simple case of chips produced by the same manufacturer, the reduction of a specific resource could prevent a design from working on a different FPGA chip. Hence designs cannot be considered portable, since a finely tuned design could not build on a different chip. By automating the design exploration process and identifying trade-offs automatically the optimisation process can be repeated for various devices without user input.

However, the issue of portability should not be restricted to the FPGA design but also viewed in the context of the entire system architecture. For example, as explained in Section 2.2.2, performance of run-time reconfiguration designs also depends on characteristics such as data transfer time and latency or whether partial reconfiguration is possible.

Integration

It is important to facilitate the integration of application code resulting from our approach with existing application code. For this it should be possible to switch seamlessly between existing software only code and dataflow based solutions. This can be achieved for example by using pragmas or providing compiler extensions that when enabled, indicate that a solution should be mapped to a dataflow accelerator and indicate how to link hardware and software.

Integration is also required between the dataflow design running on the accelerator and run-time API running on the host system. For example, in the case of parallel replication of dataflow kernels, the number of parallel processing pipelines needs to be known by both components. This concern applies to other design parameters such as word length, or run-time inputs. This suggests that there is a need to synchronise these parameters to ensure correct operation. For example the `Par` variable in the RTM kernel (Appendix B, Line 2) contains the number of parallel pipelines that are to be implemented in the design. Increasing the parallelism reduces the number of cycles for which the kernel needs to executed. Parameters sharing can be implemented via parameter files, but is simplest when both components are written in the same language.

For example using C99 syntax, FAST would be able to share configuration parameters with the CPU code (e.g. via constants or macro definitions), simplifying management and integration of the CPU and dataflow kernel.

Automation

Automating the design space exploration process results in productivity improvements. However we provide input points in our design flow which can be used gradually to tweak and guide the compilation process.

3.2 Components

To meet these requirements we propose the following approach:

- Firstly, we introduce FAST (described in Section 4), a novel language for specifying dataflow designs. We specify the accelerated portion of the original applications using FAST dataflow kernels. By maintaining compatibility with C99 syntax we improve developer productivity by providing a familiar language and introduce the possibility of combining hardware and software specifications. FAST uses simple C99 style syntax to capture the computation data path and control-path and acts as a layer on top of the MaxJ/MaxCompiler designs in our approach.
- Secondly, by using an aspect driven compilation flow we decouple optimisation from

design development, improving design portability, and we automate the generation of code and design space exploration improving productivity. We use LARA and the Harmonic aspect weaver to implement the aspect descriptions that drive the compilation process. This introduces the challenge of integrating the aspect weaver, the FAST language and our backend `fastc` compiler.

- Thirdly, systematic design space exploration is used to identify maximum performance configurations, subject to platform specific constraints. Aspect descriptions can be used to more conveniently control and guide the exploration process based on user requirements.
- Finally, fully functional MaxCompiler designs are generated from the configurations using the `fastc` compiler. This step involves automatic generation of:
  - compute kernels
  - memory access kernels
  - managers
  - run-time API calls to be inserted in the original application code

The proposed design flow is illustrated in Figure 3.1 and follows the steps:

1. a C application containing an embedded high-level dataflow design is developed from the original source application. The design is implemented using FAST as described in Chapter 4;
2. the dataflow design is transformed by the aspects in the repository to generate new designs (e.g. with multiple word-length configurations). The classes of aspects used with our approach are introduced in Chapter 5;
3. the generated configurations are compiled using a backend compilation toolchain (currently MaxCompiler) to dataflow designs implemented on FPGAs;
4. the feedback from the compilation process is used to drive the design space exploration, repeating the weaving and compilation process until user specified requirements are met.

3.3 Comparison with Existing Approaches

Compared to existing work described in [17] and [56] our approach emphasises and provides more freedom in the exploration of design level optimisation (such as word length optimisations and mapping of arithmetic blocks to DSPs) by using a combination of implementation aspects (shown in Figure 3.1) and FAST optimisation options.

Additionally, our approach targets a dataflow architecture as opposed to the von Neu-

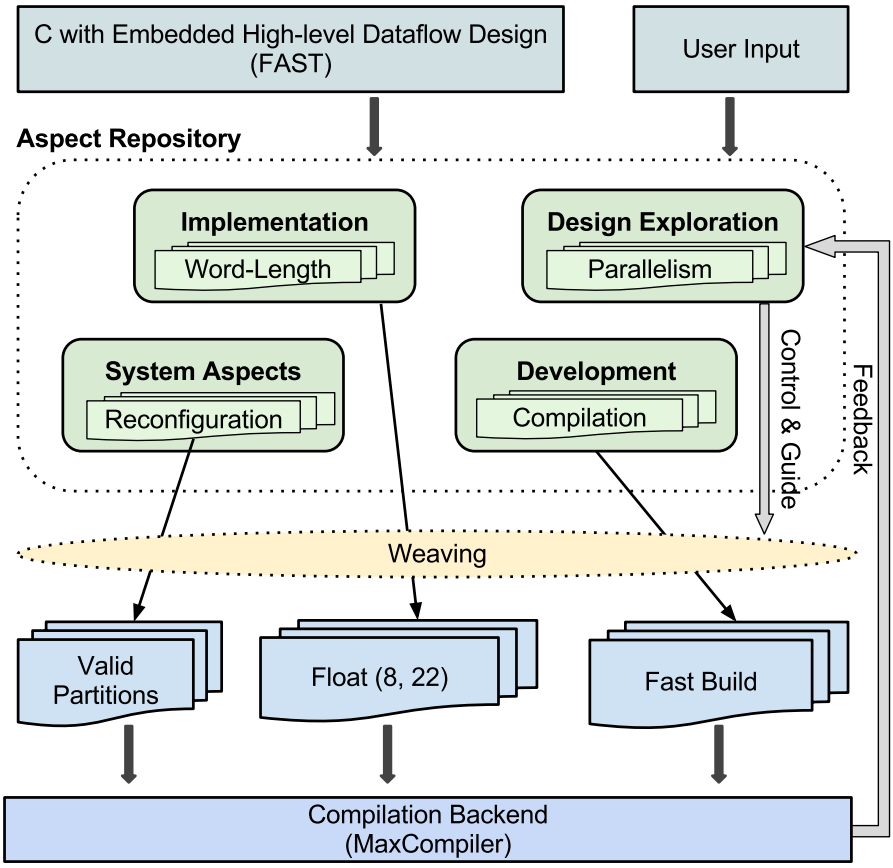

Figure 3.1: Proposed approach for aspect-driven compilation of dataflow designs.

mann architecture proposed in related work, which typically includes a General-Purpose Processor (GPP) and a custom accelerator. We consider additional optimisations to achieve performance improvements as a result of a systematic design space exploration process.

Compared to the MaxCompiler approach described in Section 2.2.3, the proposed design flow has a number of benefits, the most exciting being the capability of capturing dataflow designs using vanilla C and aspects. This is in direct contrast with the meta-programming[59] approach used in MaxCompiler which can be confusing to new users as suggested by many queries on very basic matters from novice users on the Maxeler Developer Forums[60]. In addition, the MaxCompiler design approach does not enable automating the design space exploration process effectively. Because the dataflow design and run-time component are written in different languages, integration with tools for

aspect weaving requires double the effort. The idea behind the FAST approach is to facilitate better integration with other tools by providing a single language implementation. The reason behind using meta-programming in MaxCompiler is to provide good support for design parametrisation. In the FAST approach this is achieved by specifying optimisation strategies in the aspect descriptions, simultaneously achieving the goals of decoupling optimisations form application code and simplifying the programming model for dataflow designs. This leads to improved productivity without affecting the efficiency of the generated designs.

On top of MaxCompiler, Maxeler offers a domain-specific compiler for finite difference applications, MaxGenFD[61]. This simplifies the creation of finite difference kernels (such as those required for the stencil applications described in Section 2.3). Although this abstracts effectively the creation and optimisation of stencils, the process is not transparent to the users and cannot be exposed to external design space exploration tools. Like MaxCompiler, MaxGenFD provides no specific support for designs with run-time reconfiguration.

3.4 Extensions

To support run-time reconfiguration effectively, the design flow of Figure 3.1 should be extended to support modelling and recording of design resource usage, execution time, latency and power metrics. These data are required when writing strategies for run-time reconfiguration.

3.4.1 Design Modelling

A design model for resource usage, pipeline depth and timing information can be created using limited user input. This is to be used in the run-time reconfiguration to determine optimal partition scheduling. Alternatively this information can be extracted from portable platform description files such as those used in HARNESS which contain a detailed specification of platform characteristics.

Based on this model we can estimate design performance, latency, energy efficiency and compilation time, and provide useful feedback earlier in the development process than is possible with existing tools.

Of course our model relies on user inputs for resource usage map relies on estimates for resource usage and will not be expected to provide very accurate results, especially in the presence of optimisation of backend tools and various trade-offs that can be made when mapping arithmetic to FF/LUT pairs vs DSPs.

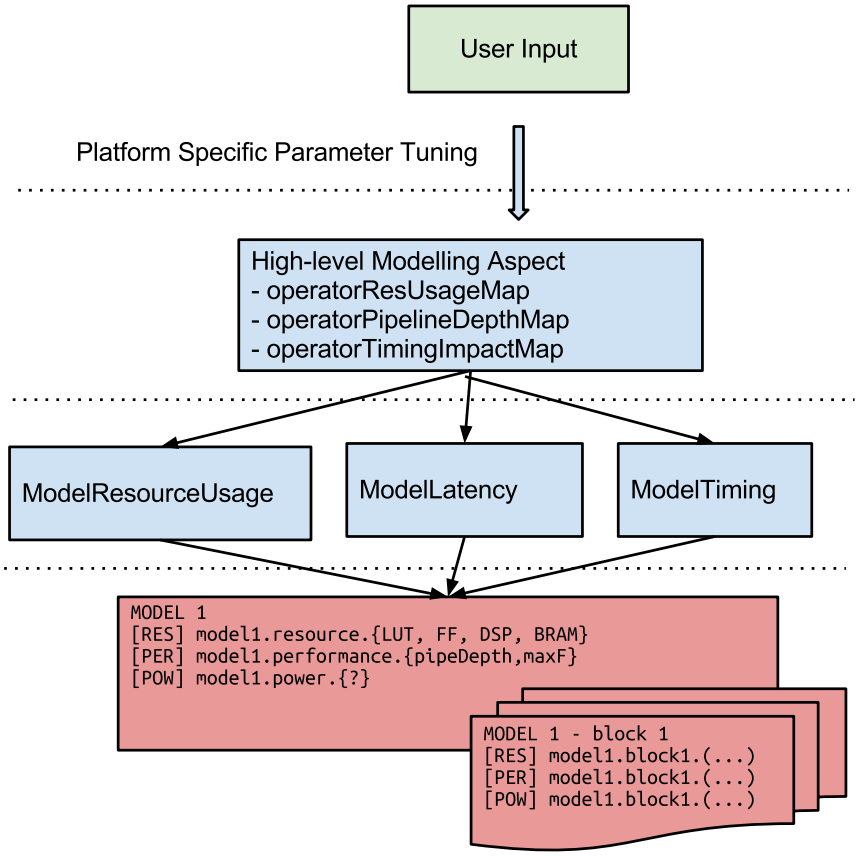

Figure 3.2: High-level aspects controlling generation of design resource, performance and timing modelling.

3.4.2 Run-time Reconfiguration Support

In the proposed design flow we distinguish between:

- **static run-time reconfiguration** – where partitions can be statically scheduled at compilation time after the design space exploration process has completed
- **adaptive run-time reconfiguration** – where partitioning depends on user input and an optimal schedule cannot be generated at compile time

Supporting both types of reconfiguration requires the introduction of an additional modelling step which is performed prior to the application of transformations for run-time reconfiguration. Additionally a performance test suite is used to measure and validate

the results of partitioning.

Figure 3.3 presents an overview of the revised design flow supporting run-time reconfiguration.

An additional requirement for adaptive run-time reconfiguration is a configuration database where generated configurations and their specification are stored for extraction, based on run-time conditions.

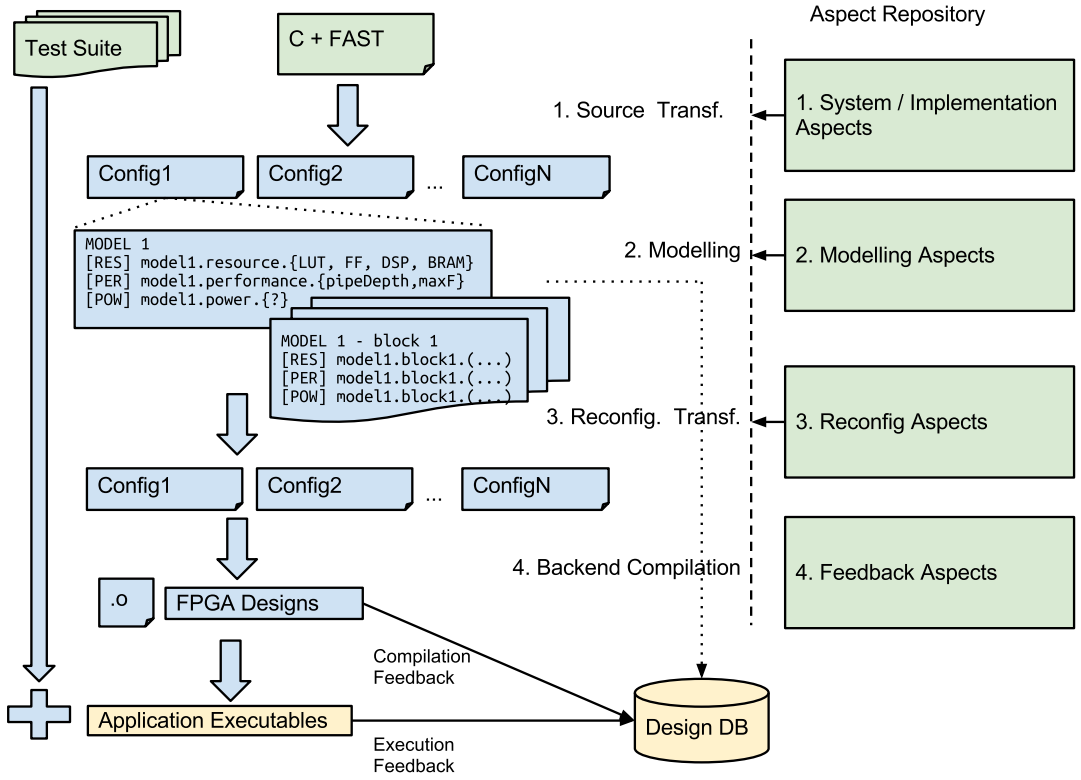

Figure 3.3: Revised design flow to support run-time reconfiguration

1. From an initial C + FAST design we create a number of configurations by applying implementation and system aspects as described in ASAP.

2. For each configuration we derive a model of the resource usage, performance and power usage using modelling aspects (described in section 3)

3. Based on predicted models (from step 2) and possibly on existing empirical information from the Design DB we partition the application and schedule the partitions to achieve user requirements and optimisation goals.

4. Finally a number of executables and designs are created, corresponding to the

generated configurations from step 3. Based on compilation and execution (of automated performance tests) we refine our predictions for design resource usage, performance and power.

Additionally the run-time environment has to be extended to support this approach. An overview of the assumed runtime architecture for the proposed approach is shown in Figure 3.4.

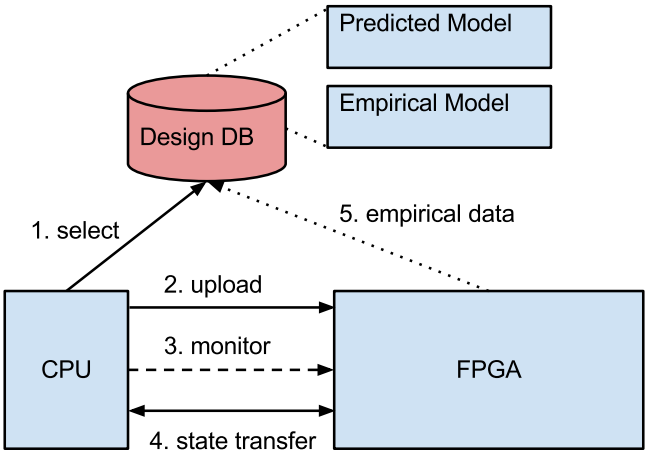

Figure 3.4: Run-time architecture required to support run-time reconfiguration

1. The CPU drives the FPGA. It initiates the computation and monitors the FPGA for factors such as temperature and power.
2. The CPU can request the FPGA to abort execution of the current design. Current results are saved (i.e. streamed back to CPU) before reconfiguration is triggered.
3. The CPU can select a new design to be uploaded
4. Empirical results are recorded in the Design Database, to improve future estimates of performance, power and thermal values.

3.5 Summary

We have introduced the proposed Aspected-oriented design flow for dataflow engines and introduced the components required to support it. Optimisations are encapsulated in aspect descriptions, separated from functional application code which leads to a more maintainable and easier to understand programming model, with minimal impact on

performance. We show what extensions are required to the original model in order to support the specification of strategies using run-time reconfiguration to increase efficiency.

Chapter 4

The FAST Language

FAST (Facile Aspect-driven Source Transformation) is a novel language for specifying dataflow designs that are used as a starting point for the design flow proposed in Section 3. In particular, we use C syntax to capture dataflow computations, and, instead of heavily relying on API libraries to specify the design (as in MaxCompiler [26] or Streams-C [54]), we use aspects to implement the transformations required for the actual implementation. In this section we outline the design goals of the language, introduce an early prototype and examine its advantages and limitations and propose an enhanced version. We highlight the main features of FAST and explain how these translate to components of dataflow designs. We analyse the major challenges we met in creating the FAST language and highlight the strategies we adopted for overcoming them.

4.1 Design Goals

FAST provides the following features that are required by the proposed flow:

- *Imperative specification of dataflow designs.* C99 syntax is enforced by the FAST compiler which is based on the ROSE Framework [62]. We have chosen C99 syntax for the following reasons:
  - the standard is widely adopted and should be instantly familiar to developers
  - it simplifies integration with many existing high-performance applications (which tend to be written in C/C++)
  - it includes a limited number of basic features (arithmetic, pointers, functions) but also some convenient escape hatches (such as macros, type definitions or pragmas) that can be used to complement these features in a standard manner
  - the Harmonic Aspect Weaver, used in the HARNESS project only supports

C99 syntax so this provides a mean to integrate our dataflow designs with the wider HARNESS project

  - considerable infrastructure is readily available (compilers, source-to-source translation and optimisation tools exist for C/C++)

- *Good integration with existing source level translation and weaving tools.* Simple syntax allows the language to interact well with existing compilers or source to source translation frameworks, allowing source level optimisations to be applied through different tools.

- *Combined hardware/software design.* Specifications of dataflow kernels and CPU run-time software can be mixed. The example shown in Listing 4.1 can be compiled with the GCC toolchain, but when using the FAST compiler, the pragma indicates the link between the software and hardware, which results in an accelerated hardware/software solution.

- *Support for data path and control path generation.* FAST allows specifying both data and control operations that are automatically mapped to stream multiplexers.

FAST is used to express the simplest form of a dataflow design while optimisations and other transformations are encapsulated in aspects which are developed separately and applied through aspect weaving. This results in a flexible approach for generating and exploring the space of efficient dataflow designs.

Designs in FAST are compiled to MaxCompiler designs composed of inter-connected functional kernels. Communication between kernels is asynchronous, so they can operate independently, and compute only when all active inputs have available data.

4.2 Features

We originally developed a simple prototype of the FAST language sufficient to support our application benchmark (Chapter 6). We then extended this prototype with more advanced features (as described in Section 4.3) which were required to meet our design goals.

In this section we use a very simple implementation of a dataflow kernel which is part of our Black Scholes benchmark to highlight some of the main features of the FAST language. This kernel simply computes the result of the finite difference approximation solution to the Black Scholes Equations, iterating both in space (the fast dimension – stock prices) and in time (the slow dimension). In Section 4.4 we analyse the same example in light of our extensions described in Section 4.3 and argue that these simplify the language.

```
1  void Price_FPGA(s_float8_24 stockPrices,
2                  float8_24 c1, float8_24 c2, float8_24 c3,
3                  int32 nStocks, int32 stencilOrder, int timesteps)
4  {
5    // read input stream
6    in(stockPrices);
7
8    // counters for timestep and stockstep iteration
9    int32 timestep  = count(timesteps, 1);
10   int32 stockstep = countChain(nStocks, 1, i4);
11
12   #pragma FAST DSPBalance:full
13   s_float8_24 result =   stockPrices[ 0] * c1
14                        + stockPrices[ 1] * c2
15                        + stockPrices[-1] * c3;
16
17   // boundary conditions for the stencil computation
18   int32 up = (stockstep >= stencilOrder)
19              && (stock < stockstep - stencilorder);
20
21   // write output stream
22   out(up ? result : stockPrices);
23 }
24
25 // Regular C style CPU implementation
26 void Price_CPU(...) {...}
27
28 int main() {
29   #pragma FAST hw:Price_FPGA
30   Price_CPU(...);
31 }
```

Listing 4.1: FAST dataflow kernel for Black Scholes Options pricing

Listing 4.1 highlights some of the most important features of a FAST dataflow kernel:

- dataflow kernels are declared as regular C functions with inputs defined as arguments in the function signature;
- streams are represented through `s_<type>` types, which are type definitions for `<type *>`; these are interpreted as special types by the `fastc` compiler;
- to provide easy access to previous, current and future stream values array notation is used with positive offsets accessing future values and negative offsets accessing previous stream values
- supported offset values are linear combinations of compile-time constants or variables (either loop induction variables, or normal variables but for which a compile time range of values is specified, as a requirement for generating efficient hardware)
- constructs such as loops are supported as long as their bounds are known at com-

pilation time and are used to parametrise dataflow designs.

- C function calls are mapped to dataflow kernels via pragmas (Line 17) which provides the flexibility of selecting a particular dataflow configuration based on run-time conditions

Additionally and API is provided for higher-level constructs such as I/O functions (`in()`, `out()`), counters (Listing 4.1, Lines 4–5) and functions to multiplex streams (`mux()`).

Table 4.1 summarises the features of FAST and shows that many features are implemented the using C99 style syntax. Mathematical functions are specified using their standard C library results in a more intuitive, easier to learn programming model but complicates the mapping of FAST designs to FPGA dataflow designs. Another important consequence of maintaining compatibility with C99 syntax is the ability to directly translate C applications to dataflow designs.

Feature	**Description**	**Method (see Listing 4.1)**
Input/Output	Declared in function header	C99 (line 1)
	`in()`,`out()`	FAST API (lines 2,11)
Control	Ternary op., `if` statement	C99 (line 11)
	Stream mux (`mux()`)	FAST API
Computation	+, *, /, -	C99 (line 8)
	log, exp, sqrt, sin etc.	#include <math.h>
Streams	Declared as pointers	C99 (line 1)
	Accessed with array index	C99 (line 8)
Optimisation	C pragmas	C99 (line 7)
Parameterization	Constants, variables	C99
Hardware Mapping	C pragmas	C99 (line 17)

Table 4.1: Summary of the main features of the FAST language.

In the remainder of this section we provide an in-depth analysis of these features and design challenges associated with capturing them in a simple imperative language.

4.2.1 Kernels and Streams

Kernels represent a unit of computation that is mapped to the FPGA. They are C functions that can be linked via pragmas placed before a corresponding CPU (non-

accelerated) function call in the C application. It is not convenient, or correct for that matter, to allow the direct calling of dataflow kernels. First of all, from the execution model point of view this would not make sense and would confuse the user since a "call" to the dataflow kernel more precisely maps to a sequence of calls where the data are streamed, one value per cycle, into the kernel and the stream counters are incremented at each kernel iteration (as shown in Algorithm 4). Secondly, this can also lead to spurious warnings and even unexpected errors when compiling and running with a different compiler than `fastc`.

Algorithm 4 Kernel Execution Loop

```
function RunKernel(kernel, cyclesToRun)
    kInParams ← kernel.InputStreamPointers
    kOutParams ← kernel.OutputStreamPointers
    kConstantParams ← kernel.ConstantParams
    MAX_CYCLE ← cyclesToRun
    for cycle ∈ 1 ... cyclesToRun do
        CURRENT_CYCLE ← cycle
        call kernel(kInParams, kConstantParams)
        for streamPointer ∈ kStreamParams do
            streamPointer ← streamPointer + 1
        end for
    end for
    for streamOutPointer ∈ kOutParams  do
        streamOutPointer ← streamOutPointer - cyclesToRun
    end for
    return kOutParams
end function
```

However, although disallowing direct calls to FAST dataflow kernels enforces a more robust separation between the dataflow and CPU components, it introduces the complication of passing parameters to the kernel. We use the convention over configuration approach [63] to simplify parameter passing, assuming the following implicit mapping:

1. the first parameter of the CPU function call maps to the number of kernel cycles; this is mapped to a special variable named `MAX_CYCLES` which is accessible within the dataflow kernel; it does not need to listed as a separate parameter in the kernel definition

2. stream and constant parameters map to the identically named dataflow kernel parameters

3. output streams are listed in identical order in the kernel function call as in the dataflow kernel design

4. for situations where these conventions are not ideal, we provide the means of speci-

fying parameter mappings using pragma parameters of the form `map:stream_CPU, stream_Kernel`.

5. the special variable `CURRENT_CYCLE` is updated with the current cycle count at each cycle as per Algorithm 4

These conventions are illustrated in Listing 4.2 and the resulting parameter mapping is explained in Table 4.2.

CPU Parameter	**Dataflow Parameter**	**Explanation**
n	MAX_CYCLES	Rule 1
x	a	Rule 4
y	y	Rule 2
y	s	Rule 2
prod	result1	Rule 3
sum	result2	Rule 3
–	CURRENT_CYCLE	Rule 5

Table 4.2: Mapping of parameters from CPU function calls to FAST dataflow kernel.

Additionally we assume that all input and output streams and constant values are correctly allocated according to the C99 standard before the function call to the dataflow kernel is occurs.

```
1  // standard C main function
2  int main() {
3    // allocate and pre-set x, y, prod and sum
4
5    #pragma fast kernel:MovingAverage map:x, a
6    MovingAverage(n, x, y, s, prod, sum);
7
8    // do something useful with prod and sum
9  }
10
11 void MovingAverage(int32 *a, int32 *y, int32 s) {
12   int32 result1 = a[0] * y[0] * CURRENT_CYCLE;
13   int32 result2 = a[0] + y[0] + s;
14   out(result1); out(result2);
15 }
```

Listing 4.2: Simple FAST dataflow kernel.

The kernel inputs can be streams or run-time constants. The kernel produces as output

one or more streams of data. We distinguish between:

- **output/write stream** – allows access to previous and current values; write streams are created via calls to the `out` function as shown in Listing 4.2;
- **input/read stream** – allows access to previous, current and future values; these are the streams defined in kernel declaration (e.g. `a` and `y` in Listing 4.2);
- **mixed output/input** streams – allows access to previous, current and future values; mixed streams are streams that are defined within the kernel body.

Stream Type

A key to obtaining high-performance FPGA designs is the use of custom data types, where the FPGA offers a higher degree of flexibility than the C implementation. We can for example create fixed point precision data types of arbitrary bit width for integer and fractional part or non-standard floating point formats. A key design space exploration step is that of identifying required data types based on application specific accuracy requirements. For example in the case of Reverse Time Migration decreasing operator width can result in dramatic improvements in performance (2 times faster) with unnoticeable effects on image quality [64].

Hence, variable bit width integer, fixed and floating point data types should be supported. However, the C standard does not provide means to specify arbitrary width types [65, pp. 33]. Although arbitrary bit width fields can be specified this can only be done inside `structs` and requires padding to a multiple of 8 bits, which severely limits and complicates the use of values defined in this manner.

An alternative to the standard defined types is the introduction of custom type definitions. These can bear specific meaning to the `fastc` compiler while remaining completely transparent to standard compilers such as GCC. This initial approach is illustrated in Listings 4.2 and 4.1. A complete list of the type definitions and their corresponding C99 types is shown in Table 4.3.

Stream Offsets

Stream offset expressions are used to access previous or future stream values using the array index notation (as shown in Listing 4.3). These expressions should be linear combinations of compile time constants or constant inputs to the kernel. More efficient hardware can be constructed if bounds for the offset are specified. This can be done via the `pragma var:var_name type:offset min:min_value max:max_value` shown on Line 1 of Listing 4.3.

The array index notation offers a simple means of accessing stream values but introduces,

FAST Type	C Type	Example	Explanation
float(exp)_(mant)	float	float8_22	Single precision floating point value with 8 exponent bits and 22 mantissa bits
double(exp)_(mant)	double	double11_50	Double precision floating point value with 11 exponent bits and 50 mantissa bits
fixed(exp)_(mant)	float	fixed3_12	Fixed precision value with 3 integer bits and 12 fractional bits
int(width)	int	int15	Integer value with 15 bits
uint(width)	int	uint15	Unsigned integer value 15 bits
s_(type)	type*	s_float8_24	Stream of single precision floating point values
s_array_(type)	type**	s_array_int32	Stream array of integer values

Table 4.3: FAST custom data type for variable bit width integer, fixed and floating-point values.

for maintaining compatibility with the C standard to annotate all stream variables. For example this leads to the superfluous `x[0]` on Line 3 of Listing 4.3. In the context of large dataflow kernels it can be tedious and error-prone to annotate stream values so we introduce the possibility of declaring stream values as regular scalar (non-pointer/array) types. For example, on line 4 of Listing 4.3, the value of r can be used without need for annotation. This, however introduces the limitation that future or past values of the stream "r" cannot be accessed in the kernel anymore, as shown on Line 4.

```
1 #pragma fast var:off type:offset min:-128 max:128
2 void (int32 *x, int32 off) {
3   int32 r = x[-off] + x[off] + x[1] + x[-1] + x[0];
4   int32 o = 2 * r + 5;
5   // int32 o = 2 * r[-1] + 5;  -- Illegal!
6   out(o);
7 }
```

Listing 4.3: FAST kernel using offsets.

4.2.2 Control and Computation

Regular control statements can be used. When conditionals are based on stream values, the control statements are mapped by `fastc` to hardware multiplexers.

It is only possible to use loops statements (while, for) if their bounds and induction variables are known at compile time. In Listing 4.4 the loop is used to generate parallel arithmetic pipelines for every pair of inputs followed by an adder tree that reduces the result.

Computation is captured using a mix of C operators and standard functions:

- C arithmetic operators can be used as usual on stream values (not streams themselves);
- C math.h function calls are automatically mapped to efficient hardware blocks;
- Arithmetic on streams (equivalent of pointer arithmetic) is illegal. Instead stream values must be extract by use of the current values operators (* or [0]).

```
1  // constants can be used for design parametrisation
2  const int nPairs = 2;
3
4  void PairwiseSquareRootSum(s_array_float8_24 x) {
5    float8_24 prod, sum;
6
7    // loop is used for design parametrisation
8    for (int i = 0; i < nPairs; i++) {
9      prod = x[2 * i][0] + x[2 * i + 1][0];
10
11     // C99  arithmetic functions can be mapped to hardware blocks
12     sum = sum + sqrt(abs(prod));
13   }
14   out(sum);
15 }
```

Listing 4.4: Compute and control example in FAST

4.2.3 Pragmas

Pragmas are used to indicate information that pertains to the optimisation process rather than the functionality of an application. In particular they indicate:

- optimisation options exposed by the backend tools
- additional type information required to generate variable width representations of operands

One exception is the hardware software linkage pragma which maps a software call to its corresponding dataflow engine version.

The use of pragmas enables users to switch seamlessly between compilers and, eventually backend compilers, contributing towards the Integration requirement of our design flow.

However, unlike in other approaches pragmas are not meant to be inserted manually – although this is possible – but rather they are to be controlled by the corresponding aspect descriptions (for hardware / software partitioning, optimisation etc.). The use of pragmas enables aspect descriptions to operate correctly (or rather to not operate incorrectly) accross various platforms (since, by definition, compiler directives are simply ignored if they are not understood by the compiler) and contributes towards our Portability requirement.

Pragams in FAST follow the syntax:

```
#pragma fast (param_name:param_value)* (func:func_name)?
```

The function name parameter simplifies loading of pragma information into a global data structure.

4.3 Extensions

In this section we present the extensions we implemented on top of our FAST and `fastc` prototype in order to improve ease of use and simplify the language as much as possible. This simplifies adoption by users and also integration with the Harmonic Aspect weaver.

4.3.1 Inferring Stream Type

Our original approach to specifying variable stream types is problematic since it fails to decouple effectively the variable bit width optimisation from the application code. This in turn can lead to complications when writing aspect descriptions for exploring the design space of bit width optimisations since a more intensive analysis of the entire application is required in order to recognise optimisation pointcuts and generate correct designs when attempting to vary the representation of certain operands.

To handle this situation we observe that, in general to classes of types are interesting to vary:

- I/O type – this is the representation of a stream that when interfacing with the CPU; this has to be a standard C type to avoid data corruption
- kernel/compute type – this the representation of a stream used inside the dataflow kernel, and hence, on the FPGA dataflow engine. This can be varied freely to

non-standard types subject to accuracy requirements

Hence we introduce the following pragma for IO `pragma fast var:stream_name ioType:typeToUseForIO computeType:typeToUseInsideKernel`. Such pragmas can be automatically inserted by the aspect weaver based on various optimisation strategies and can be easily compiled by `fastc` into corresponding MaxCompiler designs.

Additionally, the `fastc` compiler can be extended to automatically infer input and output streams (without requiring superfluous calls to the `in()` and `out`) method calls based on the following rules:

- If a stream is assigned to at least once then it is a write stream
- If a stream is assigned to more than once, then this is an error
- If a stream is declared in the kernel header and not written to, then it is a read stream
- If a stream is declared within the body of the dataflow kernel, it is a read/write stream

4.3.2 Multiple Kernel Support

Some designs may require more than one dataflow kernel. Indeed all the applications in our benchmark contain at least three kernels: one computational kernel, and two kernels for generating read and write commands for the on-board DRAM. This is a typical use case where one kernel is to perform operations asynchronously from the other: merging the command generator kernels with the computation kernel can lead to a congested design and very easily to a kernel freeze (the equivalent of a deadlock in software). Hence, a recurrent pattern is where a separate kernel is used to generate the memory command stream, which contains the addresses that are to be read from DRAM.

To support this scenario, MaxCompiler uses the concept of a manager which specifies how kernels are instantiated and connected together to form a design (as described in Section 2.2.3).

To support this scenario in FAST we extend our original pragma notations to enable the specification of:

- correlation between input and output
- kernel instances

4.4 Revised FAST Example

Listing 4.5 shows the revised Black Scholes implementation with the revised FAST language. No API calls are required for the counters or state saving. Inputs and outputs are clearly declared in the kernel header and the compiler can automatically infer whether a parameter stream is input or output. The type width information is decoupled from the application code and can be added automatically via aspect generated pragma statements.

Additionally, we can use FPGA DRAM (bandwidth of 40GB/s) as the source for data, not just PCI-Express (2GB/s) which is a major improvement in terms of I/O bandwidth. This is achieved through our DRAM extensions and the support for multiple kernel that allows us to fully specify a three kernel design that implements the required pricing computation.

```
1  // 1. both input and output streams are declared in kernel header
2  // 2. no need for additional type definitions
3  void Price_FPGA(float* stockPrices, float *r,
4                  ... /* same as before */)
5  {
6
7    // 3. CURRENT CYCLE value used instead of counter API
8    int stockstep  = (CURRENT_CYCLE / n1) \% timesteps;
9
10   #pragma fast DSPBalance:full
11   int result = ....; /* same as before */;
12
13   // 4. boolean types used for conditions
14   bool up = ...; /* same as before */ );
15
16   // 5. assigning to output stream automatically outputs value
17   r[0] = up ? result : stockPrices;
18 }
```

Listing 4.5: FAST dataflow kernel for European Options pricing

4.5 The `fastc` Compiler

The `fastc` compiler is an experimental compiler for the FAST language. It takes C applications with embedded FAST dataflow designs and produces a functionally correct and complete MaxCompiler design.

`fastc` extends the ROSE[66, 62] compiler framework to provide support for generating dataflow designs. ROSE uses EDG as its fronted to produce an Abstract Syntax

Tree(AST) from C sources. The resulting AST is directly manipulated by `fastc` in a series of compiler passes. Finally, MaxJ code is generated for the dataflow kernels and the original C is left unmodified. In addition, `fastc` produces a Make file and deployment script based on user configuration to completely automate the development of the dataflow design.

The following sequence of compiler passes is run on the input FAST + C code:

1. **Extract dataflow kernels** – separates FAST dataflow kernels from the rest of the source C application and initialises the `Design` object which is passed through to each subsequent pass. The `Design` object contains references to the extracted kernels, and is subsequently enriched with information about the structure and properties of the dataflow design which is required for the final code generation pass. Identification of kernels is performed via querying the ROSE AST for:
   - all function names beginning with `kernel_`
   - all pragma definitions of the form `#pragma fast dataflow` which are located exactly before a function definition (or after a number of other pragma declarations)
2. **Constant Extraction** – extracts design constants from the source files. These are global constants used to parameterise the FAST dataflow designs and can be shared with the CPU code. This type of parameterisation is useful for example when defining the width of an input stream (in terms of number of elements per cycle). Note that design parameters that are not constant are consider illegal since they would lead to ambiguities in the execution model (for example users might expect them to be synchronised between the CPU and dataflow components at run-time). The set of design constants is given by the kernel read only values (obtained through a read/write analysis step) minus the set of kernel par meters;
3. **Pragma Extraction** – the pragma extraction pass analyses pragmas that specify types of streams, ranges of offset streams etc. These are a pre-requisite for subsequent passes for type inference and checking and code generation. The extracted information is used to update the `Design` object;
4. **Infer input and output streams** – as explained in Section 4.3.1, `fastc` can infer the direction of streams by following the steps:
   (a) extract kernel parameter set, P
   (b) extract pointer parameters which corresponds to the stream set, $S \subset P$
   (c) perform a written analysis and record the kernel modifies set, M
   (d) compute the set of output streams, $O = M \cap S$
   (e) compute the set of inputs streams, $I = P - O$

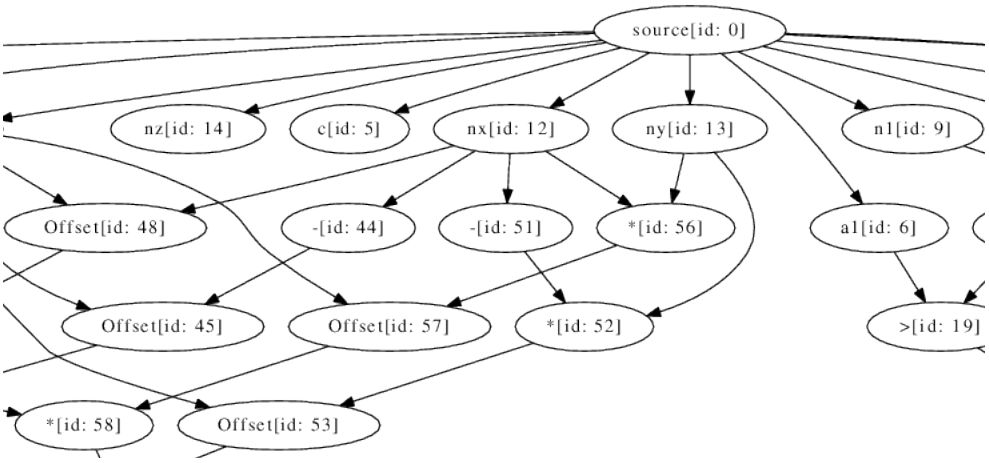

Figure 4.1: Section of a dataflow graph generated using fastc.

(f) detach assignments to output streams from the original AST to prevent traversal on future passes and add corresponding output (and input) nodes in the dataflow design

5. **Inline auxiliary functions** – perform inlining of all function calls that appear inside a FAST dataflow kernel body (except those defined as part of the language API).

6. **Static Single Assignment Renaming** – rename variables to the Static Single assignment form; this step is required after inlininig to ensure that duplicate local variable name clashes do not occur;

7. **Type Checking and Inference** – based on the type of input and output streams and values of constants infer and check type consistency; this task is simplified by the fact that the MaxCompiler backend supports some degree of type inference. A complication arises when using Boolean variables inside the

8. **Dataflow Graph Generation** – traverse the AST of every dataflow kernel to create a corresponding dataflow graph. An example generated dataflow graph is show in Figure 4.1. It illustrates `Input`, `Offset` and `Arithmetic` nodes. The in edge of input nodes is connected to a special node name `Sink`. Additional examples of `fastc` DFG nodes include `Counter` and `Output`. The out edge of output nodes is also connected to a special `Source` node;

9. **Remove FAST** – removes the function nodes corresponding to the dataflow kernels from the AST and verifies AST integrity (namely that there were no direct calls to these kernels)

10. **MaxJ Code Generation** – traverse the constructed dataflow graph and the original AST generate corresponding MaxJ Design for extracted dataflow kernels

4.6 Summary

We have introduced the FAST language one of the components required as part of the proposed Aspect-oriented design flow. We have shown the most important features of the FAST language and how they map to hardware components on the FPGA based dataflow engine. We have highlighted the challenges of capturing these constructs with a simple imperative language such as C and some possible solutions which we investigate in Section 4.5.

Table 4.4 contrasts the FAST approach proposed in this project with existing approaches in terms of syntax and programming paradigm. An imperative, as opposed to functional paradigm simplifies the language, making it more accessible to novice users and integrates well with existing application source code. The dataflow style of the FAST specifications allows for an efficient specification of designs that map well onto hardware accelerators such as FPGA-based dataflow engines. The language supports integrated hardware/software co-design with existing C applications, simplifying the design exploration process by improving sharing of parameters and by exposing a unified syntax to external design space exploration tools (such as the LARA design space exploration flow).

Finally, Table 4.5 highlights the support for design parametrisation and optimisation exploration strategies via the automated aspect-oriented design flow as opposed to manual transformations or meta-programming used by existing state-of-the art compilers.

Language	**Syntax**	**Paradigm**	**Support**
Lucid	Lucid	Functional	Software
SISAL	SISAL	Functional	
Lustre	Lustre	Synchronous	Combined
MaxCompiler	C99(SW) Java(HW)	Imperative(SW) / Dataflow(HW)	
Streams-C ImpulseC	C99	Imperative(SW) / CSP(HW)	
FAST/LARA	C99(SW/HW) LARA(Aspects)	Imperative(SW) / Dataflow(HW)	

Table 4.4: Syntax, paradigm and support comparison of the FAST/LARA approach and existing dataflow implementations.

Language	Implementation	Design Parametrisation	Optimisation Strategies
Lucid **SISAL** **Lustre** **MaxCompiler**	Multiprocessor	Manual Source Transformation Meta-programming	Manual Code Revision
Streams-C **ImpulseC**	CPU + FPGA	Compiler Directives	
FAST/LARA		Compiler Directives + Automated Aspect-Directed Source Transformation	

Table 4.5: Implementation, parametrisation and optimisations comparison of the FAST/LARA approach and existing dataflow implementations.

Chapter 5

Aspect Descriptions

Aspects descriptions are modules that capture functional cross-cutting concerns that are decoupled from the primary function of a program. In traditional Aspect-oriented approaches, program execution points (e.g. method calls) are intercepted at run-time to allow new code to be executed before, after or in place of these execution points. The process through which this is achieved is called *weaving*. The main motivation behind AOP is to solve the modularisation problem when dealing with multiple cross-cutting functional concerns.

The LARA aspect-oriented design-flow [55], depicted in Figure 5.1, performs the weaving process at compile-time to meet non-functional optimisation goals, such as improving application performance on particular hardware platforms. The weaving process manipulates and transforms the application sources generating new sources (woven code) that incorporate both functional elements of the original sources, and non-functional concerns captured by LARA aspects.

In this project we combine the LARA aspect design-flow with FAST dataflow designs. As explained in Chapter 4, FAST uses standard C99 syntax to capture dataflow computations while aspects specify decoupled optimisation and transformation strategies that operate on FAST descriptions. This approach makes the functionality of the application easier to understand, more maintainable and portable since it is no longer obscured by various structural or algorithmic transformations, or platform specific optimisations. In addition, strategies coded in LARA can be re-applied automatically in different applications, thus improving developer productivity.

We introduce novel aspect descriptions to use with FAST dataflow designs, which we group in four main classes as shown in Table 5.1:

- **System Aspect Descriptions** are applied at the whole system level (FAST + C application) to capture the mapping between application modules and accelerator

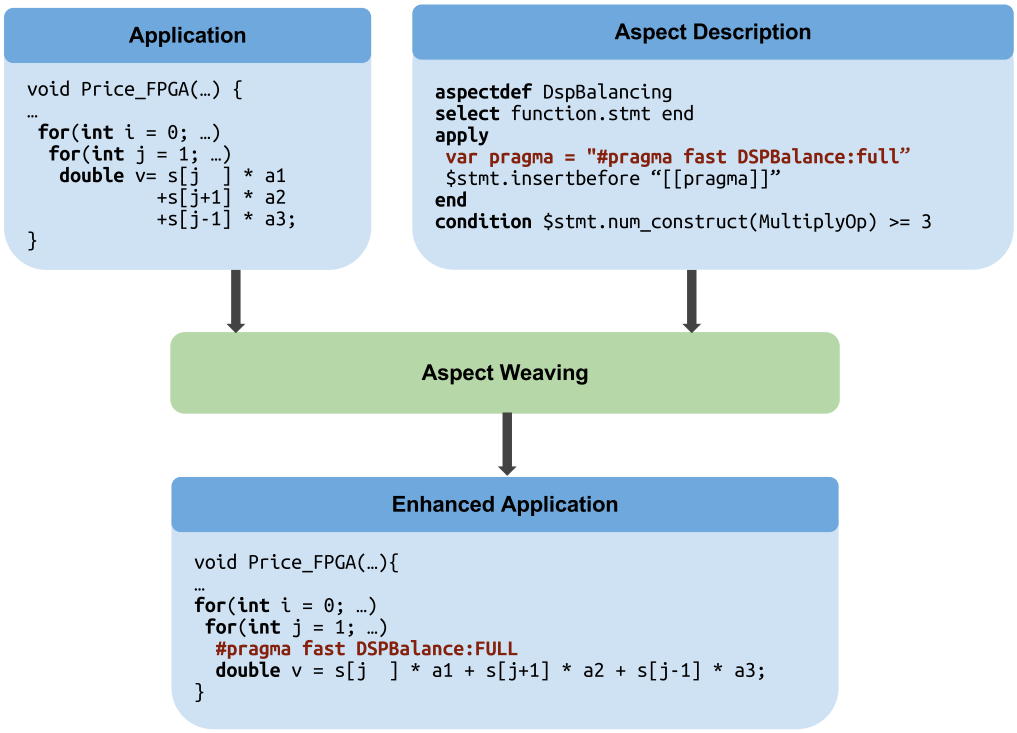

Figure 5.1: Aspect weaving overview.

- **Implementation Aspect Descriptions** are applied at the level of the dataflow design, to apply platform specific optimisations
- **Exploration Aspect Descriptions** are used to explore the design space of optimisation trade-offs
- **Development Aspect Description** are used to support the development process

5.1 System Aspect Descriptions

System aspect descriptions capture transformation or optimisation strategies that affect the whole application such as those concerning hardware/software partitioning, monitorisation and run-time reconfiguration capabilities. The goal of hardware/ software partitioning is to improve the overall execution time by identifying parts of the code to be offloaded to hardware (Section 5.1.1). Monitorisation aspects instrument the application code to extract run-time behaviour, and uncover opportunities for optimisa-

Table 5.1: Types of Aspects used in FAST

Aspect Type	Aspect Name	Description
system	• hw/sw partitioning • monitorisation • reconfiguration	capture mapping between application modules and GPP/FPGA accelerators
implementation	• operator optimisation • word-length spec	capture low-level hardware optimisations
exploration	• iterative • metaheuristic	generate multiple implementations based on design space exploration strategies
development	• simulation • debugging • compilation	improve developer productivity

tion (Section 5.1.2). Run-time reconfiguration can be used to remove idle functions from the accelerator at specific points in time, so that additional resources can be dedicated to functions that are active [3].

5.1.1 Hardware/Software Partitioning

FAST functions describing dataflow computations can be embedded within the C application but, as explained in Section 4.2.1, cannot be invoked directly by software C functions. Instead, a FAST pragma must be used to indicate the link between the software function call and the alternate hardware implementation as shown in Listing 5.1. This indicates that the software implementation of `f()` can be mapped to the dataflow implementation described in `fast_f()`. This way, our design-flow can automatically switch from a pure software application to a software/hardware design.

```
1 void fast_f()  {/* dataflow implementation */}
2 void      f()  {/* software implementation */}
3 int main() {
4   #pragma fast hw:fast_f
5   f();
6 }
```

Listing 5.1: Mapping of C function calls to dataflow kernels using FAST pragmas.

Hence, a hardware/software partitioning strategy can be performed in five steps:

1. detecting hotspots in the program;
2. detecting code patterns from hotspots that are suited for dataflow computation and acceleration;
3. performing the outlining transformation so that each candidate for acceleration is enclosed in a function `f`;
4. deriving a dataflow version `fast_f` from state-based `f`;
5. placing a FAST pragma on top of each function call to `f` and associate it to the corresponding `fast_f` function.

Each of these steps can be described as a separate LARA aspect and combined to form a hardware/software partitioning strategy.

5.1.2 Monitorisation

To find potential hotspots in the application, we can use the aspect in Listing 5.2. With this aspect, the weaver can automatically instrument any C application to self-monitor its innermost loops at run-time, as they are natural candidates for dataflow-based acceleration. In particular, this monitorization aspect can compute the following information for every innermost loop:

1. the average number of times it has been executed,
2. the average number of iterations,
3. the loop average execution time,
4. the loop iteration average execution time.

For this purpose, we use a simple monitoring API, to record the frequency of execution and the loop iteration and execution time:.

- `monitor_instanceI` and `monitor_instanceE` – mark the beginning and end of the loop respectively;
- `monitor_iterI` and `monitor_iterE` – mark the beginning and end of an iteration respectively.

```
1 aspectdef LoopMonitor
2 select function.loop{is_innermost} end
3 apply
4     $loop.insert before
5         %{monitor_instanceI("[[$loop.key]]");}%;
6     $loop.insert after
7         %{monitor_instanceE("[[$loop.key]]");}%;
```

```
8  end
9
10 select function.loop{is_innermost}.entry end
11 apply $begin.insert after
12       %{monitor_iterI("[[$loop.key]]");}%;
13 end
14 select function.loop{is_innermost}.exit end
15 apply $begin.insert before
16       %{monitor_iterE("[[$loop.key]]");}%;
17 end
18 end
```

Listing 5.2: Aspect that instruments the application to monitor loop activity. The information generated can be used to identify hotspots.

The aspect code is shown in Listing 5.2. Lines 2–8 select all innermost loop instances and place an instance monitor call before and after each selected loop. Lines 10–13 select all entry points inside the loop and insert a monitoring call to mark the beginning of each iteration. Lines 14–17 place an instance monitor call to mark the end of each iteration. The following table shows an example of applying the aspect from Listing 5.2 on a C-style function containing a loop:

original code	**woven code**
`void f() {`	`void f() {`
	`monitor_instanceI("f:1");`
`  while (i < N) {`	`  while (i < N) {`
	`  monitor_iterI("f:1");`
`  i++;`	`  i++;`
	`  monitor_iterE("f:1");`
`  }`	`  }`
	`monitor_instanceE("f:1");`
`}`	`}`

Each monitoring call in the woven code receives as a parameter a unique loop key which identifies the loop within the application. The loop key is generated by concatenating the function name with the hierarchical position of the loop within the abstract syntax tree. For instance, *f:2:1* corresponds to the 1st loop inside the 2nd outermost loop of function `f`. The hotspots can be identified by an aspect (not shown) that takes the profiling information generated by the monitorization API calls, and that uses an heuristic to compute the most profitable computations to be offloaded to hardware.

5.1.3 Modelling

A prerequisite of developing more advanced aspects is the ability to effectively model resource usage. In practice, an approximate resource usage and performance model is developed by application engineers, prior to the design optimisation process. Presently, this is done manually, or can be automated but not at a high enough level abstraction (e.g. C programming level). An automatically derived resource model improve developer productivity by providing design information earlier in the development process, albeit just an estimates.

Modelling aspect descriptions can be used to generate design models for FAST dataflow designs. For example, the aspect description of Listing 5.3 can be used to estimate resource usage of arithmetic operations. It relies on the existence of an operation resource usage map (opUsageMap) that contains an estimate of the per operation resource usage. This estimate can be refined via analysis of backend tool reports, using the feedback capabilities of the LARA design space exploration flow. Based on this estimate the aspect updates the current designModel with estimates of total resource usage for each loop loop statement

```
1  aspectdef usageLUT
2  input
3    opUsageMap, designModel
4  end
5    var totalUsageLUT = 0;
6  select function.statement
7  apply
8    for(var op in {+, -, *, /})
9      nOpps = statement.num_construct[op]
10     usageLUT = opUsageMap[op].LUT * nadds * loop max iteration
11     switch (loop.balanceDSP) {
12       FULL: usageLUT = 0
13       BALANCED: usageLUT *= 0.5
14       NONE: usageLUT *= 1
15     }
16     desginModel.block[loop].resource.LUT = usageLUT
17     totalUsageLUT += usageLUT
18     designModel.resource.LUT= totalUsageLUT
19   end
20 end
```

Listing 5.3: Aspect for modelling resource usage of arithmetic operations.

To model design performance we can, for example, use the aspect of Listing 5.4 to estimate the pipeline depth, based on which design latency can be estimated as depth $*$ design clock frequency. This operates similarly to the previous aspect, by expecting as input an operator pipeline depth and then estimating the pipeline depth of loop

constructs.

```
1  aspectdef pipelineDepth
2  input
3    opPipelineMap, designModel
4  end
5  var designDepth = 0;
6  select function.loop{is_innermost}
7  apply
8    loopDepth = tree_depth(loop_body, opPipelineMap) * loop.pipelineFactor
9    designDepth += loopDepth
10   designModel.block[loop].performance.pipeDepth = loopDepth
11   designModel.performance.pipeDepth = designDepth
12   designModel.performance.pipeDepth = designDepth
13 end
```

Listing 5.4: Aspect for modelling pipeline depth of arithmetic kernels.

These aspects are called by a higher level aspect that is intended as the primary means for users to interact and parametrize the lower level aspects. This hierarchical approach ensures a gradual abstraction of details, which simplifies the development process. The aspect of Listing 5.5 provides a single entry point for the modelling flow, where the operator can maintain a list of platform level properties. Such portable descriptions already exist for more widely used FPGA based systems, requiring just a simple translation step to fully automate the modelling process.

```
1  aspectdef designModel
2  input
3  opPipelineMap
4  opUsageMap
5  end
6  for (var design in DesignDatabse)
7    call usageLUT(opUsageMap, design)
8    call pipelineDepth(opPipelineMap, design)
9  end
```

Listing 5.5: Higher-level aspect for modelling design resource usage.

5.1.4 Run-time Reconfiguration

Run-time reconfiguration aspects are used to partition and generate hardware/software links required for reconfiguration based on user specified requirements and optimisation goals.

To support run-time reconfiguration, we specify the configuration associated with the

function call in the FAST pragma. For instance:

```
#pragma FAST hw:fast_f0 cfg:c0
x = f(0);
#pragma FAST hw:fast_f1 cfg:c1
y = f(x);
#pragma FAST hw:fast_g cfg:c1
z = g(x);
```

With the above code annotations, our design-flow can generate multiple configurations, each containing a set of FAST implementations that can be executed in parallel. If the configuration name is not specified using the FAST pragma, then we assume a default configuration. Having a single configuration can lead to situations where at any point in time and due to data dependencies, part of the functions are idle. With run-time reconfiguration, we can exploit unused resources to support active functions. In particular, during the execution of an application, we select various configurations at different points in time to maximise the utilisation of FPGA resources. Within this context, we use a hardware partition, which contains a set of configurations that are used to support reconfiguration during the life cycle of an application. In the above example, configuration `c0` contains a single implementation of `f` (`fast_f0`), and thus can potentially use more resources and be faster than the `fast_f1` version which must share the same configuration (`c1`) with `fast_g`.

The work in [3] proposes an approach for extracting valid and efficient hardware partitions. To realize run-time reconfiguration without modifying the original code we use the aspect shown in Fig. 5.2. The input to the aspect is a hardware partition (lines 2–4). The partition is implemented as a hash table that maps a function call (key) to a hardware implementation, represented as a tuple containing the hardware implementation name (hw) and the associated configuration (cfg).

Table 5.2 shows an example of a hash table representing a hardware partition. The key (e.g. main:f:1) identifies a function call in the application, and is formed by concatenating the name of the caller function (main), the name of the invoked function (e.g. f) and a unique number (1). Line 5 in the aspect shown in Fig. 5.2 selects all function calls, and for each call found in the input partition (line 7), we set the appropriate pragma on top of the call statement (lines 10–12). We can now realize and experiment different reconfiguration designs by invoking this aspect with different hardware partitions.

5.2 Implementation Aspect Descriptions

Implementation aspects focus on low level design optimisations that can be applied to designs in FAST to improve timing or resource usage. For instance, operator optimisation

```
1  aspectdef AspReconfig
2  input
3     partition
4  end
5  select function.call end
6  apply
7     if ($call.key in partition) {
8        var cfg = partition[$call.key].cfg;
9        var hw = partition[$call.key].hw;
10       $call.insert before %{
11          #pragma FAST hw:[[hw]] cfg:[[cfg]]
12       }%;
13    }
14 end
15 end
```

Figure 5.2: Reconfiguration aspect.

Table 5.2: An example of a hardware partition, represented as a hash table, used with the reconfiguration aspect (Fig. 5.2)

partition		
$call.key	**hw**	**cfg**
main:f:1	fast_f0	c0
main:f:2	fast_f1	c1
main:g:3	fast_g	c1

aspects (Section 5.2.1) can be used to map operators in the program to dedicated hardware resources. Word-length aspects specify the numerical representation of variables and expressions in the design.

5.2.1 Operator Optimisation

To provide architectural details to FAST designs, such as mapping operators to DSP blocks, we can use the FAST pragma shown in Fig. 5.3 at the top of a statement (including code blocks). The balancing parameter corresponds to the degree of utilisation of DSP blocks in a statement.

The aspect shown in Fig. 5.4 is the strategy for balancing DSP blocks in every statement of an application. Instead of adding the above pragma manually, we provide a set of rules (lines 3–4) that define where to place the `balanceDSP` pragma. In this example,

```
#pragma FAST balanceDSP:balanced
{
  x = x * y;
  x++;
}
```

Figure 5.3: The FAST balancing pragma provides fine grained control over the mapping of computation to either DSPs or LUT/FF pairs.

we established the rule that full DSP block utilisation is applied to any statement that has 5 or more multipliers and adders, balanced if 3 or more multipliers, and no DSP utilisation otherwise.

```
aspectdef DspBalancing
var op_granularity =
 [{DspBalance:'full',MultiplyOp: 5,AddOp: 5 },
  {DspBalance:'balanced',MultiplyOp:3}];

select function.statement end
apply
   for (var i in op_granularity) {
      var gprofile = op_granularity[i];
      var match = true;
      for (var k in gprofile) {
         if (k != 'DspBalance') {
            match &= ($statement.num_construct(k)
                     >= gprofile[k]);}}
      if (match) {
         var pragma = '#pragma FAST balanceDSP:'
                     + gprofile.DspFactor;
         $statement.insert before "[[pragma]]";
         break;}}
   end
end
```

Figure 5.4: Aspect for exploring mapping of computation to DSP blocks.

5.3 Exploration Aspect Descriptions

Exploration aspects deal with strategies that generate multiple designs to find an optimal implementation based on user requirements. Exploration aspects can act on any

level of the design flow (C code, C and FAST, or FAST functions). They enable systematic exploration of trade-offs and optimisation opportunities. Examples of exploration aspects include iterative aspects (Section 5.3.1) which generate a sequence of solutions until a termination criterion is satisfied, and metaheuristic-based aspects to find optimal solutions in a very large design space.

5.3.1 Iterative Design Space Exploration

Using LARA we can implement and combine these aspects to enable systematic design space exploration of all the optimisation options exposed by the FAST backend resulting in the generation of a large number of designs. The feedback-directed compilation process of LARA can be used to capture and extract feedback from the backend reports pertaining to resource usage or timing information and automatically adjust the compilation process.

An example of a LARA aspect for design space exploration is shown in Fig. 5.5. It highlights the feedback capabilities of the design flow: the aspect will generate and build the FAST designs until the resource usage passes a specified LUT threshold, and at each step increasing a particular design attribute, such as exponent, mantissa or the parallelism of the design (by replicating the computational pipeline).

```
1  aspectdef DesignExploration
2  input
3     attribute,
4     start, step,
5     lut_threshold,
6     config
7  end
8  config[attribute] = start;
9  var i = 0;
10 do {
11   var designName = genName(config);
12   call genFAST(designName, config);
13   buildFAST(designName);
14   config[attribute] += step; i++;
15 } while (@hw[designName].lut < lut_threshold
16         && i < LIMIT);
17 end
```

Figure 5.5: Exploration aspect that generates multiple FAST designs by varying a design attribute (e.g. number of kernels or mantissa) until a LUT threshold is reached.

5.4 Development Aspect Descriptions

Development aspects capture transformations that have an impact on the development process such as debugging (Section 5.4.1), and, potentially, simulating kernels or improving compilation speed. Separating these concerns makes the original code easier to maintain and enables the automatic application of these transformations to a wide range of designs, improving developer productivity. Simulation aspects could be applied to dataflow designs to generate equivalent state-based C code thus enabling pure software simulation. Compilation aspects, on the other hand, could be applied during the development process to create versions of the dataflow design that compile faster by reducing the operating frequency, removing debug blocks or applying design-level optimisations that can resolve timing constraints. Naturally, reducing the compilation time would increase developer productivity.

5.4.1 Debugging Aspect

Because the current execution model does not provide run-time debugging of hardware designs, the easiest solution to debug designs is to log the values of various streams during execution. The insertion of debug statements can be encapsulated in aspects. It is particularly important to separate debug aspects from the original application code since debug blocks can influence the compilation time and timing constraints as well as the behaviour of the design. As an example, the aspect in Fig. 5.6 instruments the code to log every change in the value of a variable.

```
1  aspectdef WatchVar
2  select function.vref end
3  apply
4      $vref.parent_stmt.insert before
5       %{ log("[[$vref.name]]", [[$vref.name]]); }%
6      $vref.parent_stmt.insert after
7       %{ log("[[$vref.name]]", [[$vref.name]]); }%
8  end
9  condition $vref.is_out end
10 end
```

Figure 5.6: Aspect for automatically instrumenting the code to watch any change in the value of a program variable.

5.5 Summary

We have introduced the four main classes of novel aspect descriptions used with our the proposed design flow. System aspect descriptions operate at the whole system level (C + FAST specification) to generate hardware/software partitioning, run-time reconfiguration designs, monitorisation and modelling of resource usage and performance data. Implementation aspect descriptions operate at the design level and perform low level optimisations such as mapping of computation to DSPs or adjusting word length. Exploration aspect descriptions are used to drive the design space exploration process, for example, iteratively increase a certain design property (such as parallelism or clock frequency) until a certain requirement is met (e.g. resource usage threshold is exceeded or execution time requirement is met). Finally, development aspect descriptions can be used to capture strategies that are related to the development process, such as the need for monitoring values or reducing compilation time.

Chapter 6

Evaluation

We evaluate our approach by implementing a number of dataflow kernels for applications including advanced high-performance applications. We measure productivity, in terms of lines of code, number of function calls and cyclomatic complexity. We measure efficiency in terms of performance and energy consumption and compare the software only version, the version accelerated using MaxCompiler (manually written dataflow kernels) and the proposed approach (MaxCompiler dataflow kernels automatically generated using `fastc`).

Our benchmark includes a variety of interesting real-life applications, which are discussed in more detail in the remainder of this chapter:

- **Numerical Differentiation** is a 1D stencil computation which is highly sensitive to floating point accuracy and tests the variable bit width optimisation capabilities of the proposed design flow. Numerical differentiation is used in situations where the analytical form of a functions is not available, for example on large sets of experimental data;
- **Black Scholes Option Pricing** is a one dimensional stencil computation, with multiple time step iterations which requires the use of on-board DRAM to maximise efficiency; the Black Scholes model is one of the most commonly used models in finance for pricing derivatives of European options;
- **Reverse Time Migration** is an advanced high-performance application for seismic imaging. It is the most widely used application in the Oil and Gas industry for identifying geological structures that resemble patches of oil. This application is also part of the HARNESS validation studies;
- **Bitonic Sorting Network** is a high-throughput sorting network for limited input size. It can be used as the first stage of a multi-gigabyte FPGA sorting algorithm, to separate the inputs into sorted buckets of up to 256 elements, after which a

merging stage is applied. The sorting network is a recursively defined design, that heavily relies on compile-time loops, auxiliary function calls and input groups to generate a readable and parameterised description. Additionally by adapting operator bit-width to run-time inputs the network input width can be increased substantially,leading to improved throughput;

- **Add Prediction** is an application for click-through rate prediction used in Microsoft's Bing search engine for Sponsored Search. This is an arithmetic intensive kernel, and requires the ability to effectively tune operator bit width and design parameters to reach reach timing closure. This is also part of the HARNESS validation studies.

All dataflow designs are run on Maxeler MaxWorkstation [67] which comprises:

- Vectis Dataflow Engine, 24 GB DRAM, Xilinx Virtex 6 FPGA Chip
- Intel(R) Core(TM) i7 CPU 870 @ 2.93GHz, cache size 8192 KB
- 16 GB DRAM connected to the CPU
- DFEs connect to CPU via PCI Express gen2 x8

Theoretical memory bandwidth of the on-board DFE DRAM is 38 GB/s and PCI-Express bandwidth (used for transferring data from the DRAM connected to the CPU to the on-board DFE DRAM) is 2GB/s.

The process of placing and routing a dataflow design can take anywhere from 20 minutes to several days. Since this is a stochastic process, usually mutliple builds are started in parallel and when one terminates successfully the whole build process is stop and that design is used. Hence the build process requires substantial amount of DRAM and CPU cores, particulary during the design space exploration step where multiple instances of the same design are compiled to identify maximum performance configuration. Hence the builds were ran on the Custom Computing cluster machines:

- `cccad3` – 2 8 core, hyperthreaded (16 threads) Intel Xeon E5-2650 CPUs at 2.00GHz, 20MB cache, 189 GB DRAM
- `cccad2` – 2 6 core, hyperthreaded (12 threads) Intel Xeon X5650 CPUs at 2.67GHz, 12 MB cache, 94 GB DRAM
- `cccad1` – 2 6 core, hyperthreaded (12 threads) Intel Xeon X5650 CPUs at 2.67GHz, 12 MB cache, 118 GB DRAM

All CPU applications are compiled using GCC 4.4 with all optimisations enabled (-O3 flag) and FPGA Designs are compiled with MaxCompiler 2012.1.

6.1 Numerical Differentiation

Numerical differentiation is an important application in engineering and can be used to estimate derivative values when an analytical form of the function is not available. Consider for example the case of measuring a sample of displacements form which we want to derive the instantaneous speed. As explained in Section 2.3.1 a 5 point linear stencil can be used to approximate the value of the derivative.

However, in practice, experimental data often contains noise (unwanted random addition to a signal) which impacts the accuracy of the estimation. To filter the noise, a smoothing step is applied to the experimental data with the goal of improving the signal-to-noise ratio. One of the most commonly used examples is the Savitzky-Golay Filter [68] which applies a polynomial regression to a set of a m data points, also a linear stencil computation. Using higher order stencils (up to a few hundreds even) results in an smoother, less noisy regression.

Hence our implementation of the differentiation algorithm consists of two steps: *1)* applying the smoothing filter to the input data, *2)* estimating the derivative using a linear stencil. The algorithm for an arbitrary stencil order is shown in Algorithm 5.

Algorithm 5 Savitzky-Golay Numerical Differentiation

```
function NumericalDifferentiation(values, Order, sCoefs, dCoefs, sn, dn, Step)
    smoothValues ← Convolve(values, sCoefs, Order, sn)
    diffValues ← Convolve(values, dCoefs, Order, dn * step)
    return diffValues
end function
function Convolve(values, coeffs, Order, Normalizer)
    result[] ← 0
    for x = Order → (nCoefs − Order) do
        for c = 1 → nCoefs do
            result[x] ← result[x] + value[x - Order + c] * coeffs[c]
        end for
        result[x] ← result[x] / Normalizer
    end for
    return result
end function
```

Our FAST dataflow designs consist of two kernels one for smoothing and one for differentiation. We explore the possibility of generating an efficient run-time reconfigurable design either for improving design performance, or for supporting larger dimension stencils, via time-sharing. We measure resource usage and performance for stencil size of 5, 7 and 9 points and estimate the resource usage for larger stencils to identify sizes at which run-time reconfiguration becomes convenient. To maximise performance we write a parametrised design which can be parallelised up to the point where it becomes

memory bound.

Since the design uses PCI-E the maximum parallelism that can be achieved before it becomes I/O bound is given by:

$$\frac{\text{PCI-E bits per cycle}}{\text{kernel input bits}} = \frac{128}{32} = 4$$

Hence, when using PCI-E a kernel replication factor of 4 is ideal for maximising throughput.

If data were available straight from FPGA DRAM the design parallelism could be increased to:

$$\frac{\text{memory bits per cycle}}{\text{kernel input bits}} = \frac{1536}{32} = 48$$

However, for this algorithm transferring data to on-board DRAM will not improve overall throughput since data are only used once, hence we investigate the PCI-E design.

The measured resource usage scales linearly with the parallelism level as shown in Table 6.1 and shows that for a 7 point stencil, approximately 10 pipelines can mapped onto the FPGA. However, beyond 80% resource usage level, the design usually becomes fairly congested and fails to route or takes an extremely large time to achieve timing closure.

Pipes	LUT Usage	FF Usage	DSP Usage	BRAM Usage
1	10.01	6.65	8.45	0.75
2	18.91	13.21	17.51	1.50
4	37.13	25.32	33.12	3.75
6	56.17	39.28	50.31	4.50
8	79.63	51.22	49.55	7.75

Table 6.1: Pipeline scalability of the numerical differentiation algorithm for a 7 point stencil.

Table 6.2 shows that the maximum achievable stencil width with the static design (which uses both kernels onto the FPGA chip) is 29 whereas using run-time reconfiguration to swap the individual kernels increases the maximum stencil width to around 59 points (computed an assumed 4 parallel pipelines as shown on lines 4, 8 and 9 of Table 6.2).

The FAST dataflow kernel that implements the differentiation operation is shown in Section D.1. It uses no API (non-user defined) function calls and a total of 20 lines of code, compared to the original MaxCompiler design which requires 14 API calls and 37 lines of code.

Kernel	Stencil Width	LUT Usage	FF Usage	DSP Usage	BRAM Usage
GSDiff	5	2.03	1.55	0.99	0
	7	2.79	2.08	2.92	
	9	3.33	2.48	3.76	0
	$\text{Max} = 100/3.76 * 9/4 = 239/4 = 59$				
GSSmooth	5	2.34	1.44	2.23	0
	7	2.91	1.81	2.97	0
	9	3.47	2.21	3.91	0
	$\text{Max} = 100/3.91 * 9/4 = 239/4 = 57$				
Both	$\text{Max} = 100/(3.91 + 3.76) * 9/4 = 29$				

Table 6.2: Resource usage per stencil width, per kernel, per compute pipe.

6.2 Black Scholes

The finite difference implementation for the Black Scholes application is interesting since it introduces an element characteristic to stencil computations and also to designs that perform well on the Maxeler platform: multiple time step iteration. This enables and requires local data reuse via on-board DRAM to achieve maximum performance. This is because the PCI-E bus can only provide a maximum bandwidth of 2GB/s whereas DRAM achieves a maximum of 38GB/s.

The FAST kernel for the Black Scholes finite difference kernel was introduced in Section 4.4. The DRAM command read generator is shown in Listing 6.1. This requires a more complicated chain counter structure (where one counter is only enabled when its child in the chain is just about to wrap to zero) in order to control the memory access pattern. Additionally, a call to `DRAMOutput` is required to send the memory commands to the appropriate stream controller. For this reason, this kernel is an example where our approach is not very effective at reducing code size or number of API calls. Still, the equivalent MaxCompiler design shown in Appendix C requires 23 API calls and 19 lines of code, compared to 3 API calls and 13 lines of code required in FAST.

```
1 #include "fastc/fast.h"
2
3 void kernel_Cmdread(unsigned int iniBursts, unsigned int totalBursts,
4                     unsigned int wordsPerBurst, bool Enable)
5 {
6     int wordCount = count_p(32, wordsPerBurst, 1, Enable);
```

```
7       int* wrap;
8       *wrap = (wordCount == wordsPerBurst - 1) & Enable;
9       int burstCount = count_p(32, totalBursts, Burst_inc, wrap);
10      int *Control;
11      *Control = (wordCount == 0) & Enable;
12      DRAMOutput("dram_read", Control,
13                 burstCount + iniBursts,
14                 Burst_inc, 1, 0, 0);
15  }
```

Listing 6.1: FAST Memory Controller Kernel

Design space exploration using the iterative design exploration aspect shows that the design can easily fit up to 60 parallel processing pipelines. However the Figure 6.1 shows that the maximum measured parallelism is 48, since after this point the kernel becomes I/O bound even with DRAM.

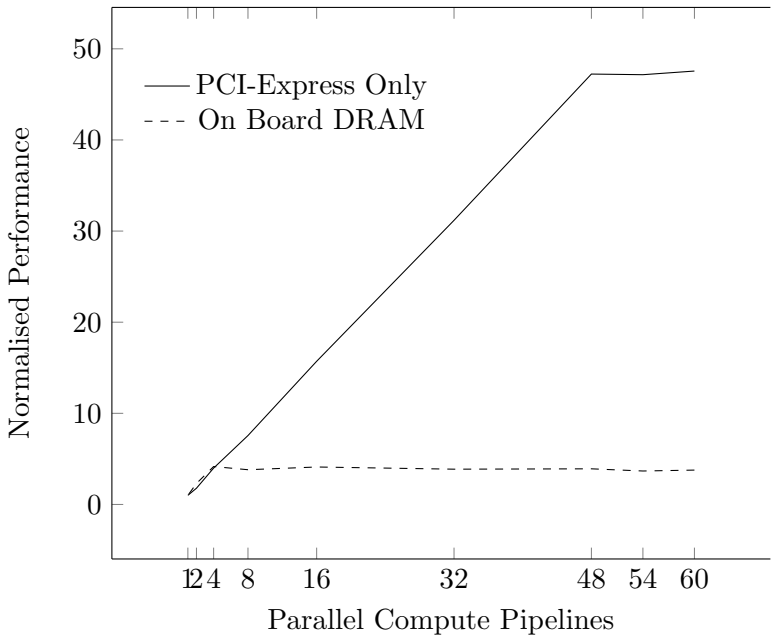

Figure 6.1: Bandwidth / computation ratio exploration using the iterative exploration aspect description.

6.3 Reverse Time Migration

The Reverse Time Migration method for seismic imaging which is used to detect geological structures, based on the Earth's response to injected acoustic waves. Background

on istropoic acoustic modelling and the RTM algorithm is introduced in Section 2.3.3. For the purpose of our implementation we approximate the differential equation using stencil computation to perform a fifth-order Taylor expansion in space and first-order expansion in time.

We use FAST to implement the dataflow kernels for both read and write memory controllers (`kernel_CmdRead`, `kernel_CmdWrite`) and the application compute kernel (`kernel_RTM`). To illustrate the benefits of our approach we analyse the results of using the debugging aspect of Section 5.4.1. Table 6.3 compares the number of lines of code required for the FAST with aspect design with the equivalent MaxCompiler implementation showing a reduction in code size of up to 42% for the run-time reconfigurable design and a reduction in the number of API calls (including debug calls) of up to 67% which translate to increased productivity.

Kernel	**Aspect**	**FAST**		**MaxCompiler**	
	LOC	**LOC**	**# API calls**	**LOC**	**#API Calls**
CmdRead	12	26	6	59	39
CmdWrite	12	28	39	79	56
RTM Static	12	246	43	403	175
RTM RTR	12	377	91	669	275

Table 6.3: Code measures for the RTM kernels comparing FAST and MaxCompiler.

Results of the design space exploration using the aspect in Figure 5.5 with variable mantissa illustrate the trade-offs between accuracy and resource usage (Figure 6.2). We observe irregular, large variations when decreasing the mantissa from 18 to 16 and 24 to 22 which is the effect of the backend tools mapping arithmetic to a combination of both DSPs and LUT/FF elements. The mantissa boundaries at which this optimisation occurs are platform specific, depending on the architecture of the DSPs. Hence, automating this optimisation via aspects and decoupling it from the original source code makes the application more portable and facilitates discovery of interesting trade-off opportunities using design space exploration.

Computation precision using floating point types can be estimated by:

$$\text{precision} = \frac{1}{2^{\text{mantissa bits}}}$$

The DSP balancing aspect shown in Fig. 5.4 allows to explore the resource trade-offs of implementing arithmetic operations in either DSPs or LUTs and FFs (Fig. 6.3) and helps to avoid over mapping on DSPs for arithmetic intensive applications.

Design space exploration using the aspect in Fig. 5.5 with increasing parallelism level

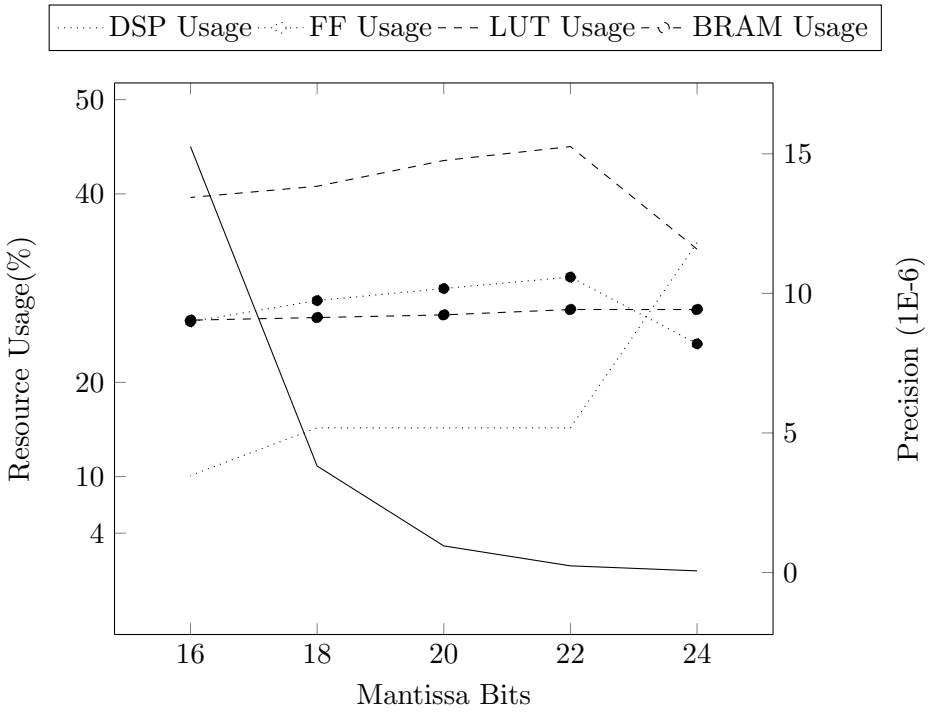

Figure 6.2: Exploration of accuracy vs resource usage trade-offs using the aspect shown in Figure 5.5 with variable mantissa.

Exp.	Mant.	FF	BRAM	LUT	DSP	Precision $(1E-6)$
8	24	24.09	27.73	34.13	34.82	0.0596
8	22	31.17	27.73	45.02	15.18	0.2384
8	20	29.96	27.16	43.54	15.18	0.9536
8	18	28.68	26.88	40.81	15.18	3.8146
8	16	26.44	26.60	39.62	10.12	15.2585

Table 6.4: Resource usage vs accuracy trade-off exploration data.

can be used to investigate design scalability. For example, for the described RTM implementation, Fig. 6.4 shows that performance scales linearly with the number of parallel pipelines and that significant speedups can be obtained by the FAST dataflow design compared to the CPU only implementation. Depending on the problem size, our approach can be used to achieve a significant speedup over software only versions which is comparable with the best published FPGA results for static designs [3], [30].

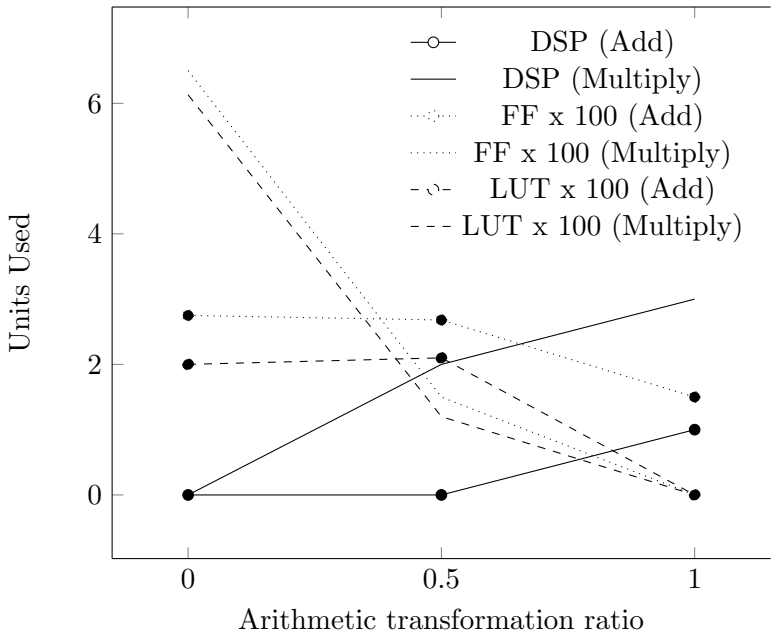

Figure 6.3: Exploration of DSP and LUT/FF balancing for functional units implementing a single arithmetic operation using the aspect shown in Fig. 5.4.

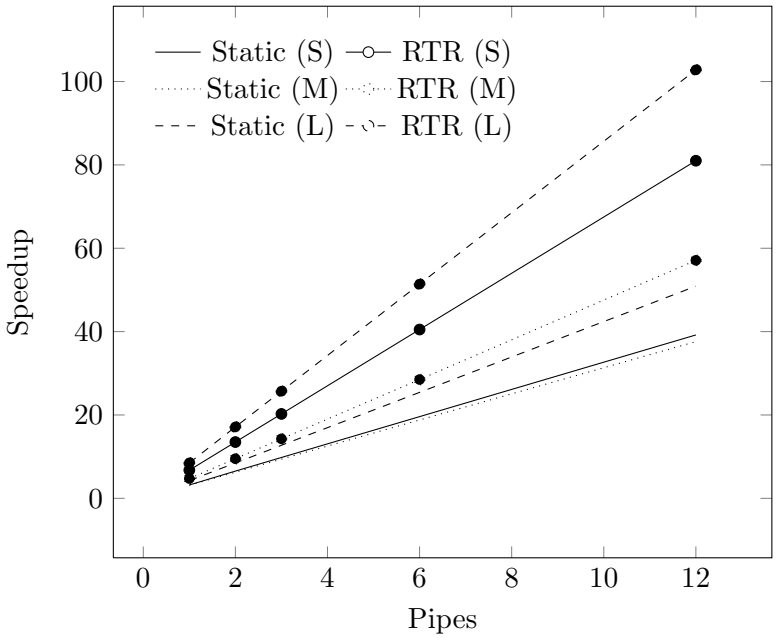

Figure 6.4: Scalability of the RTM dataflow design explored using the aspect shown in Fig. 5.5.

Fig. 6.4 also shows a model of the performance benefits of using a run-time reconfigurable implementation generated using the proposed aspect-oriented approach. Two configurations were created for the RTM FAST kernel. Since, in our model, during the first half of the execution time, the backward propagation and imaging functions are idle, the first configuration requires only half the resources. Hence, the number of parallel pipelines can be doubled, halving the execution time of the first configuration. The speedup obtained is comparable to [3], but the partitioning and optimisation exploration process is automated via aspects, which increases developer productivity. The automated process improves portability of the design, allowing optimisations based on design space exploration to be carried out on various platforms (hence subject to varying resource constraints) without manual intervention.

6.4 Bitonic Sort

Sorting networks [69] are an interesting benchmark application for our approach since it is both an important application but also fairly challenging to fit into the proposed programming model:

- Sorting networks constitute the basic blocks for high-performance multi-gigabyte sorting which in the context of the HARNESS project, is an important case study for key cloud applications;
- Sorting networks are not easily mapped to FPGA since resource constraints limited considerably the input size of the network; for example, our sorting network implementation fails to achieve timing closure
- Comparison based sorting requires very little arithmetic, which is a major disadvantage on the Maxeler Platform, since DSPs cannot be utilised
- Sorting in general is an application that does not map well onto the streaming model of computation, since due to the aforementioned resource constraints, merging of buckets of values is required which leads to a feed-back loop in the design;

We implement a bitonic sorting network for inputs of n arrays of size $k = (4, 8, 32, 64, 128, 256)$ elements. We compare the performance of the design with the ANSI C implementation of `qsort()` [1] which is a hybrid of insertion-sort and quicksort. For a fixed network size, we vary the input size to compare the software and hardware implementations and report the average results of 50 runs.

The implemented sorting network has the following properties:

- complexity = network depth = $\frac{n*log(n)}{2} = O(log^2(n))$
- comparators = $n * log(n) * (log(n) + 1)/4 = O(n * log^2(n))$

[1] http://www.umcs.maine.edu/ chaw/200801/capstone/n/qsort.c

Execution results are depicted in Figure 6.5 (experimental data is shown in Appendix E) and show that the hardware version outperforms the software version for values of n larger than 2^{14} with speedups increasing from 1.25X to 24X, in proportion with the value of n and also depending on the network input width. Thus higher speedups can be obtained for large network sizes, providing the incentive for adaptive run-time reconfiguration, based on the observed range of input values, at run-time. Although the compute only speedup is significant even for smaller values of n the execution time is dominated by the overhead of data transfer over PCI-Express from main memory to the FPGA accelerator.

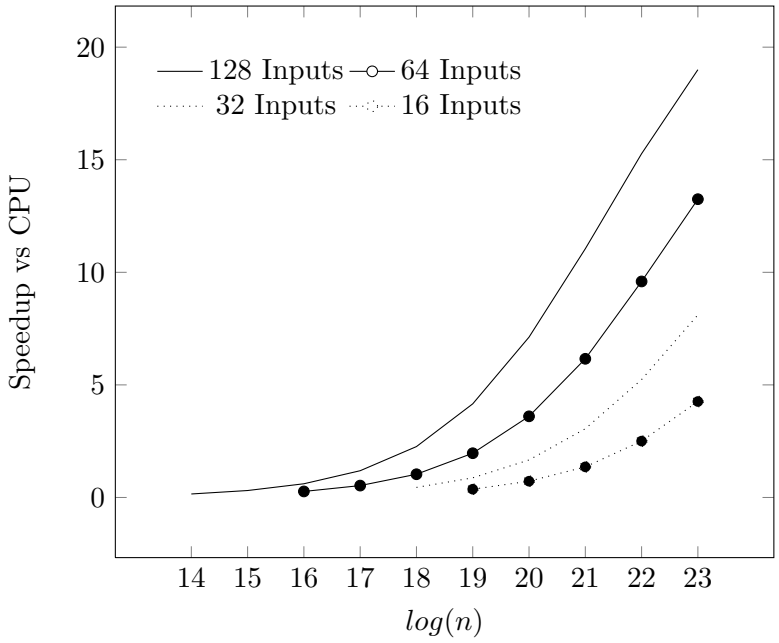

Figure 6.5: Speedup vs CPU of the bitonic sorting network designs for large batches of small inputs.

Given that execution time is dominated by the transfer time, reconfiguring the design to increase parallelism will bring a small performance benefit. The situation changes when the network is used as part of a general purpose sorting algorithm. The complexity ($O(N/k * logn * log(n/k))$ decreases linearly with the increase in the network width. This would provide the motivation to adapt the network to input patterns, switching to larger networks for small observed values. One factor to consider is that, reducing to smaller network sizes can also decrease the communication overhead. Since all network inputs must be present, additional padding is required for arrays that are not of a length which is a power of two.

Hence it is only necessary to reconfigure when a change in the input pattern is detected that requires smaller computational ranges of values. Results of exploring maximum word length using the iterative design space exploration aspect description show that

decreasing word width and varying the type (floating or fixed point, based on the input characteristics) allow us to build larger network sizes, which are capable of substantially higher throughput. Since throughput increases linearly with the networks size, reconfiguring the FPGA to adapt to smaller data representations can more than double the performance of networks for small input values. The maximum network size is determined by iteratively increasing network width until the design overmaps or fails to meet timing at the next iteration (columns 6 and 7 of Table 6.5).

Type	**Width**	**Max. Size**	**LUT %**	**FF %**	**Overmaps**	**Meets Timing**
int	32	128	42.87	43.75	yes	no
int	16	128	42.87	43.75	no	no
int	8	256	32.06	31.62	no	no
float	(8, 24)	64	34.19	18.12	yes	no
fixed	(8, 24)	128	42.79	43.75	yes	no
fixed	(24, 8)	128	42.79	43.75	yes	no

Table 6.5: Results of exploring different network sizes and data types for the bitonic sorting network

6.5 Add Prediction

The Add prediction kernel implemented for this benchmark application was proposed for predicting click-through rate for sponsored search on the Bing search engine [70]. Given as input the prior probability for a number of features, Bayesian inference is used to determine the posterior probability. The values of features are arbitrarily large (so fixed point optimisations cannot be used) and prediction accuracy increases with number of features considered by the algorithm. Increasing the number of features results in replicating most of the computational pipeline and additional levels being added to the adder tree that reduces the results. It is an interesting application because it makes use of expensive arithmetic operations such as exponentiation and square root. The original MaxCompiler implementation was developed as part of the HARNESS validation studies. Additionally, helper functions require function in lining. The results of the design space exploration show that this is a particularly challenging application to map onto the FPGA and requires ability to tune floating point mantissa. This makes use of our pragma for specifying stream I/O and compute type.

The FAST dataflow implementation is shown in Section D.2. The total number of lines of code for the Add Prediction kernel is 67 and the kernel requires 3 API calls. The

Features	DSP Factor	Representation	Meets Timing
10	Full	float(8, 24)	No
10	Balanced	float(8, 24)	No
10	Zero	float(8, 24)	No
10	None	float(8, 16)	No
10	None	float(8, 10)	No
10	Full	float(8, 10)	Yes
8	None	float(8, 24)	No
8	Full	float(8, 24)	No
8	Balanced	float(8, 24)	No
4	Full	float(8, 24)	Yes
4	Balanced	float(8, 24)	Yes
4	Full	float(8, 16)	Yes
4	Full	float(8, 10)	Yes
2	None	float(8, 24)	Yes
2	Balanced	float(8, 24)	Yes
2	Full	float(8, 24)	Yes

Table 6.6: Design exploration space for the Add Prediction kernel.

original MaxCompiler design has 90 lines of code and 49 API calls.

The iterative exploration aspect description and the operator optimisation aspect can be used for design space exploration to vary the number of features of the algorithm and the DSP mapping factor to achieve timing closure.

Table 6.6 shows the parameters used for design space exploration to achieve timing closure. The ability to map operations from LUT/FFs to DSP enables the design space exploration process to achieve timing closure.

6.6 Summary

We have analysed our implementation on a number of real-life applications. Tableconc:summary summarises our experimental results and shows that for the considered applications we can achieve close to identical performance to hand crafted MaxCompiler designs, while requiring significantly less line of code and API calls even without taking into account savings that can be achieved from using aspect descriptions.

Our extensions for multiple kernel support and supporting designs with DRAM has

Kernel	**LOC Ratio**	**API Calls Ratio**	**Performance***	**Resource***
CmdRead	1.76	4.33		
CmdWrite	1.45	4.13		
RTM	1.17	10		
SGSmooth	1.85	14	Identical	Identical
SGDifff	1.75	14		
Black-Scholes	2.5	5.5		
Add Prediction	67	16		

Table 6.7: Lines of code, API calls performance and resource usage ratio of original manual MaxCompiler design and FAST design.

helped match the performance of MaxCompiler designs that benefit from higher memory bandwidth (such as Black Scholes or RTM). In the meantime the ability to infer types has helped simplify the language reducing the number of API calls substantially. The aspect oriented design flow can be used to effectively explore the design space of available optimisation for operand bit width, or varying specific design constants in order to achieve timing closure or maximise performance. Although these results are promising, it must be noted that these metrics alone are not a definitive indication of increased productivity and that future large scale studies should be performed.

One limitation of our current implementation is that we cannot control design parameters that are not directly accessible to the dataflow kernel. For example stream clock frequency and DRAM frequency are controlled from within the MaxCompiler manager. To work around this limitation for evaluation purposes we have reproduced the same manager design in both the original MaxCompiler design and the FAST design when measuring performance.

Chapter 7

Conclusion

We have tackled the challenge of improving developer productivity with minimal impact on efficiency for custom dataflow designs implemented on FPGAs. We have shown that this can be achieved by adopting the Aspect-oriented design philosophy of encapsulating cross-cutting concerns (such as optimisations) in highly cohesive aspect descriptions.

7.1 Summary of Achievements

We introduced a novel development approach for dataflow designs, required to integrate, for the first time the Aspect-oriented approach with dataflow design development for FPGAs. By decoupling optimisations from design specification the proposed design flow both simplifies the development and maintenance of dataflow applications and highlights opportunities for platform specific optimisations.

To support the proposed design flow we introduced FAST a novel language for specifying dataflow designs which facilitates integration with existing aspect weaving tools by adopting a standard C99 syntax. Other important features of FAST include support for hardware/software co-design, which allows embedding of dataflow kernels in regular C style applications and support for variable bit width operand representation via a pragma based mechanism. We implemented a compiler for the FAST language that translates FAST dataflow designs to MaxCompiler 2012.1 designs based on the MaxCompilerRT interface. To complement the FAST dataflow designs we introduced a number of novel aspect descriptions that enable effective design space exploration with minimal user input.

We evaluated our approach on a number of applications and showed that significant improvements can be achieved in terms of productivity at minimal cost to performance. We have shown FAST dataflow designs require significantly less API calls and lines of code, while matching the performance of manually created MaxCompiler designs and

improving flexibility which simplifies the design space exploration process. Th proposed flow can be used to support design space exploration that highlights interesting trade-off opportunities for increasing design parallelism or overall throughput subject to required accuracy.

A full paper based on this project was accepted for publication at the 24th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP 2013. Finally, the project has been included in the FP7[1] funded HARNESS Project where it is to be used for generating efficient dataflow implementations for key cloud applications based on user requirements.

7.2 Future Work

Although we have met our original objectives set out in Section 1.2, we would like to highlight that substantial work remains to be done both to improve the quality of existing work (by improving for example error handling, documentation and extensibility of our compiler prototype) and to investigate the applicability of our approach to exciting classes of problems. Current and future work possibilities include:

- *Extending the approach to cover other classes of parallel computation.* Based on the positive evaluation results, we believe that the approach can be extended to support the generation of efficient dataflow designs for additional classes of parallel computation such as Sparse or Dense Linear Algebra, MapReduce or N-Body Simulation which are quintessential examples of parallel kernels [71]. This could also provide valuable feedback which can be used to refine and validate the proposed approach.

- *Extension to cover heterogeneous systems.* With the increasing demand in cloud computing solutions, it is believed that heterogeneous computing platforms can provide a better mix of performance, energy and cost efficiency than traditional CPU only based platforms. From the onset of this project we have kept the approach as platform agnostic as possible, to enable the support for reconfiguration our approach is intentionally platform agnostic to allow extension to other platforms for dataflow computing (such as CPU and GPGPUs).

- *Support a standardised set of pragams.* In the context of heterogeneous computing platforms, to facilitate the adoption of the FAST language a standardised set of pragmas such as OpenACC could be supported. We have not started with this idea in order to maximise the flexibility of our approach, but we believe that a standardised set of pragmas can help to improve portability and interoperability of C + FAST applications.

[1] European Union Seventh Framework Programme

- *Support MaxCompiler 2013.1.* MaxCompiler 2013.1 introduces a new interface and interesting opportunities for optimisations, which are highly relevant to the cloud model of shared compute resources. For example, it introduces the possibility to control groups of dataflow engines and it provides improved support for remote operation via Remote DMA over high speed Ininiband connection. These features make it interesting, if not mandatory, to support the more recent version of MaxCompiler.

- The proposed design flow can be extended *to support additional languages.* This does require a better separation in the `fastc` between the dataflow specific components, which would serve as common backend representation for interfacing with MaxCompiler and the language specific front-end passes.

- *Extension to cover other applications domains.* Based on the previous extension, domain specific languages for application domains ranging from Monte-Carlo simulations in finance [72] to genetic sequence matching [73] could be supported.

- Current *support for run-time facilities* is limited and more work is required to implement the run-time inter-facing between the CPU application and the dataflow designs. One of the reasons for postponing the implementation of run-time support was the imminent move to the MaxCompiler 2013.1 interface, which as mentioned previously introduces a heavily revised CPU – DFE interface.

- An interesting side-effect of our approach is that maintaining strict compatibility with the C99 syntax simplifies *translation from regular C / C++* applications to FAST dataflow designs. Hence FAST itself could be used as an intermediate language, a target for the translation process.

- Although based on theoretical improvements and results observed from our evaluation suite, we have all the reasons to believe that our approach can improve developer productivity and portability of dataflow designs a thorough study should must be carried out to *validate our claim of improved productivity and portability.* This was beyond the scope of this project since it required a level of effort which could not have been achieved within the limited time span;

- Aspect descriptions can be used to *support verification* of designs, thus simplifying what is often a more elaborate process than design development itself. For example, the proposed approach is compatible with verification by symbolic simulation and equivalence checking as explained in [74];

- Finally, the implementation presented in this report is a prototype and, although functional, it requires *further engineering* work before being available for release to the public. In particular, error handling is not very robust and the extensibility of the compiler could be improved, to allow open programmatic access to the dataflow graph representation generated by the `fastc` compiler.

Bibliography

[1] M. Flynn, O. Pell, and O. Mencer, "Dataflow Supercomputing," in *FPL*, 2012.

[2] O. Mencer, "Maximum Performance Computing for Exascale Applications," in *SAMOS*, 2012.

[3] X. Niu, Q. Jin, W. Luk, Q. Liu, and O. Pell, "Exploiting Run-Time Reconfiguration in Stencil Computation," in *FPL*, 2012.

[4] Tiobe Software, "Tiobe Programming Index," http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2012.

[5] Y. Lam, J. Coutinho, and W. Luk, "Integrated Hardware/Software Codesign for Heterogeneous Computing Systems," in *SPL*, 2008.

[6] J. Cardoso, P. Diniz, and M. Weinhardt, "Compiling for Reconfigurable Computing: A Survey," *ACM Computing Surveys (CSUR)*, vol. 42, no. 4, p. 13, 2010.

[7] A. V. Aho, J. D. Ullman, and S. Biswas, *Principles of Compiler Design.* Addison-Wesley, 1977.

[8] M. Wolfe, C. Shanklin, and L. Ortega, *High Performance Compilers for Parallel Computing.* Addison-Wesley Longman, 1995.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming," *ECOOP*, 1997.

[10] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K. Hill *et al.*, "Exascale Software Study: Software Challenges in Extreme Scale Systems," *DARPA IPTO, Air Force Research Labs, Tech. Rep*, 2009.

[11] D. Chen, J. Cong, and P. Pan, *FPGA Design Automation: A Survey.* "Foundations and Trends in Electronic Design Automation", 2006.

[12] Q. Jin, T. Becker, W. Luk, and D. Thomas, "Optimising Explicit Finite Difference Option Pricing for Dynamic Constant Reconfiguration," in *FPL*, 2012.

[13] S. Weston, J.-T. Marin, J. Spooner, O. Pell, and O. Mencer, "Accelerating the Computation of Portfolios of Tranched Credit Derivatives," in *WHPCF*, 2010.

[14] O. Pell, J. Bower, R. Dimond, O. Mencer, and M. Flynn, "Finite Difference Wave Propagation Modeling on Special Purpose Dataflow Machines," *IEEE Transactions on Parallel and Distributed Systems*, 2012.

[15] D. Oriato, S. Tilbury, M. Marrocu, and G. Pusceddu, "Acceleration of a Meteorological Limited Area Model with Dataflow Engines," in *SAAHPC*, 2012.

[16] J. Dongarra, G. Fagg, A. Geist, J. Kohl, P. Papadopoulos, S. Scott, V. Sunderam, and M. Magliardi, "HARNESS: Heterogeneous Adaptable Reconfigurable NEtworked SystemS," in *HPDC*, 1998.

[17] J. Cardoso, J. Teixeira, J. Alves, R. Nobre, P. Diniz, J. Coutinho, and W. Luk, "Specifying Compiler Strategies for FPGA-based Systems," in *FCCM*, 2012.

[18] O. Pell and O. Mencer, "Surviving the end of Frequency Scaling with Reconfigurable Dataflow Computing," *SIGARCH Comput. Archit. News*, pp. 60–65, 2011.

[19] P. Grigoras, X. Niu, J. G. Coutinho, and W. Luk, "Aspect Driven Compilation for Dataflow Designs," in *ASAP (to appear)*, 2013.

[20] O. Lindtjrn, R. G. Clapp, O. Pell, O. Mencer, and M. J. Flynn, "Surviving the End of Scaling of Traditional Micro Processors in HPC." IEEE HOT CHIPS 22, 2010.

[21] S. Trimberger, "Scheduling designs into a time-multiplexed fpga," in *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. ACM, 1998, pp. 153–160.

[22] D. Koch and J. Torresen, "Fpgasort: a high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2011, pp. 45–54.

[23] R. Marcelino, H. C. Neto, and J. M. Cardoso, "Unbalanced fifo sorting for fpga-based systems," in *Electronics, Circuits, and Systems, 2009. ICECS 2009. 16th IEEE International Conference on*. IEEE, 2009, pp. 431–434.

[24] Maxeler, "MaxCompiler white paper." [Online]. Available: http://www.maxeler.com/media/documents/MaxelerWhitePaperMaxCompiler.pdf

[25] Xilinx, "Virtex 6 family overview." [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf

[26] O. Lindtjorn, R. G. Clapp, O. Pell, O. Mencer, M. J. Flynn, and H. Fu, "Beyond Traditional Microprocessors for Geoscience High-Performance Computing Applications," *Micro, IEEE*, vol. 31, no. 2, pp. 41–49, 2011.

[27] J. N. Lyness and C. B. Moler, "Numerical differentiation of analytic functions," *SIAM Journal on Numerical Analysis*, vol. 4, no. 2, pp. 202–210, 1967.

[28] planetmath.org, "Solving the Black-Scholes PDE by finite differences." [Online]. Available: http://planetmath.org/solvingtheblackscholespdebyfinitedifferences

[29] E. Baysal, D. D. Kosloff, and J. W. Sherwood, "Reverse Time Migration," *Geophysics*, vol. 48, no. 11, pp. 1514–1524, 1983.

[30] M. Araya-Polo *et al*, "Assessing Accelerator-based HPC Reverse Time Migration," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 147–162, 2011.

[31] W. L. Xinyu Niu, Qiwei Jin and S. Weston, "A Self-Aware Tuning and Evaluation Method for Finite-Difference Applications in Reconfigurable Systems," 2013.

[32] G. Kiczales, "Aspect-oriented Programming," in *ICSE*, 2005.

[33] I. Jacobson and P.-W. Ng, *Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2004.

[34] J. Gradecki and N. Lesiecki, *Mastering AspectJ: Aspect-Oriented Programming in Java*. Wiley, 2003.

[35] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "An overview of AspectJ," *ECOOP 2001Object-Oriented Programming*, pp. 327–354, 2001.

[36] O. Spinczyk, A. Gal, and W. Schröder-Preikschat, "AspectC++: an Aspect-Oriented Extension to the C++ Programming Language," in *TOOLS Pacific*, 2002.

[37] J. Cardoso, K. Bertels, G. Kuzmanov, R. Nane, and V. Sima, "REFLECT: Rendering FPGAs to Multi-Core Embedded Computing," *Reconfigurable Computing: From FPGAs to Hardware/Software Codesign*, pp. 261–289, 2011.

[38] J. M. P. Cardoso, T. Carvalho, J. Teixeira, P. C. Diniz, F. Goncalves, and Z. Petrov, "Hardware/Software Specialization Through Aspects: The LARA Approach," in

SAMOS, 2012.

[39] M. G. Corp, "Catapult C synthesis C to hardware concepts," October 2009.

[40] R. Ferrer, J. Planas, P. Bellens, A. Duran, M. Gonzalez, X. Martorell, R. Badia, E. Ayguade, and J. Labarta, "Optimizing the exploitation of multicore processors and GPUs with OpenMP and OpenCL," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, K. Cooper, J. Mellor-Crummey, and V. Sarkar, Eds. Springer Berlin Heidelberg, 2011, vol. 6548, pp. 215–229. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19595-2_15

[41] "Online C to verilog translator." [Online]. Available: http://www.c-to-verilog.com/online.html

[42] Xilinx, "Vivado design suite." [Online]. Available: http://www.xilinx.com/products/design-tools/vivado/index.htm

[43] "Impulse C." [Online]. Available: http://www.impulseaccelerated.com/products_universal.htm

[44] S. Czerniawski, "Building Deep, Hazard-free Hardware Pipelines from OpenCL Programs," Master's thesis, Imperial College London, 2011.

[45] Khronos, "OpenCL specification." [Online]. Available: http://www.khronos.org/opencl/

[46] E. A. Ashcroft and W. W. Wadge, "Lucid, a Nonprocedural Language with Iteration," *Communications of the ACM*, vol. 20, no. 7, pp. 519–526, 1977.

[47] B. W. Tony Faustini, "pLucid." [Online]. Available: http://www.gnu.org/software/automake/manual/html_node/Autotools-Introduction.html

[48] J. Gurd and W. Bohm, "Implicit Parallel Processing: SISAL on the Manchester Dataflow Computer," *Proceedings of the IBM-Europe Institute on Parallel Professing*, 1987.

[49] J. McGraw *et al.*, "SISAL: Streams and Iteration in a Single-assignment Language," Lawrence Livermore National Lab, Tech. Rep., 1983.

[50] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The Synchronous Data Flow Programming Language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[51] N. Halbwachs, F. Lagnier, and C. Ratel, "Programming and Verifying Real-time

Systems by Means of the Synchronous Data-flow Language LUSTRE," *IEEE Transactions on Software Engineering*, vol. 18, no. 9, pp. 785–793, 1992.

[52] V. Vijayaraghavan, K. Kavi, and B. Shirazi, "Control flow extensions to the dataflow language sisal," in *Applied Computing, 1991., [Proceedings of the 1991] Symposium on*, apr 1991, pp. 130 –138.

[53] K. Kavi, V. Vijayaraghavan, B. Shirazi, and A. Hurson, "Barriers and break-points in dataflow: extensions to sisal language," in *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on*, vol. i, jan 1992, pp. 526 –534 vol.1.

[54] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented FPGA Computing in the Streams-C High Level Language," in *FCCM*, 2000.

[55] J. M. Cardoso, T. Carvalho, J. G. Coutinho, W. Luk, R. Nobre, P. Diniz, and Z. Petrov, "LARA: an Aspect-Oriented Programming Language for Embedded Systems," in *AOSD*, 2012.

[56] J. M. Cardoso, R. Nane, P. C. Diniz, Z. Petrov, K. Krátkỳ, K. Bertels, M. Hübner, F. Gonçalves, J. G. d. F. Coutinho, G. Constantinides *et al.*, "A New Approach to Control and Guide the Mapping of Computations to FPGAs," in *ERSA*, 2011.

[57] J. Cardoso, "Programming Strategies for Runtime Adaptability," in *ReCoSoC*, 2012.

[58] D. H. Jones, A. Powell, C.-S. Bouganis, and P. Y. Cheung, "Gpu versus fpga for high productivity computing," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*. IEEE, 2010, pp. 119–124.

[59] T. Sheard, *Accomplishments and Research Challenges in Meta-programming*, ser. Lecture Notes in Computer Science, W. Taha, Ed. Springer Berlin Heidelberg, 2001, vol. 2196.

[60] Maxeler, "Maxeler developer exchange." [Online]. Available: https://groups.google.com/a/maxeler.com/forum/?fromgroups#!forum/mdx

[61] MaxelerFD, "MaxGenFD white paper." [Online]. Available: http://www.maxeler.com/media/documents/MaxelerWhitePaperMaxGenFD.pdf

[62] D. Quinlan, "ROSE: Compiler Support For Object-Oriented Frameworks," *Parallel Processing Letters*, vol. 10, pp. 215–226, 2000.

[63] N. Chen, "Convention over configuration," 2006. [Online]. Available: http:

//softwareengineering.vazexqi.com/files/pattern.html

[64] H. Fu, "Accelerating scientific computing through gpus and fpgas."

[65] "C99 language standard." [Online]. Available: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf

[66] L. L. N. Laboratory, "ROSE compiler infrastructure." [Online]. Available: http://www.rosecompiler.org

[67] Maxeler, "Maxeler maxworkstation specification." [Online]. Available: http://www.maxeler.com/products/desktop/

[68] A. Savitzky and M. J. Golay, "Smoothing and differentiation of data by simplified least squares procedures." *Analytical chemistry*, vol. 36, no. 8, pp. 1627–1639, 1964.

[69] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. ACM, 1968, pp. 307–314.

[70] T. Graepel, J. Q. Candela, T. Borchert, and R. Herbrich, "Web-scale bayesian click-through rate prediction for sponsored search advertising in microsofts bing search engine," in *Proceedings of the 27th international conference on machine learning*, 2010, pp. 13–20.

[71] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," EECS Department, University of California, Berkeley, Tech. Rep., 2006.

[72] Q. Jin, D. Dong, A. Tse, G. Chow, D. Thomas, W. Luk, and S. Weston, "Multi-level Customisation Framework for Curve Based Monte Carlo Financial Simulations," in *ARC*, 2012.

[73] J. Arram, K. Tsoi, W. Luk, and P. Jiang, "Hardware Acceleration of Genetic Sequence Alignment," in *ARC*, 2013.

[74] T. Todman and W. Luk, "Verification of streaming designs by combining symbolic simulation and equivalence checking," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. IEEE, 2012, pp. 203–208.

[75] GNU, "Autotools introduction." [Online]. Available: http://www.gnu.org/software/automake/manual/html_node/Autotools-Introduction.html

[76] Boost, "Boost C++ Library." [Online]. Available: http://www.boost.org/users/

history/version_1_47_0.html

Appendix A

User Guide

`fastc` is an experimental compiler for the FAST language and is available at `https://github.com/paul-g/maxcc`.

Installation

The project requires [1]: *1)* GNU AutoTools [75], *2)* Boost 1.47[76] and *3)* ROSE 0.9.5 [66]. To extract and compile the project please run the following commands:

```
1 tar xvzf fastc-${version}.tar.gz && cd fastc-${version}
2 configure --with-boost=/path/to/boost --with-rose=/path/to/rose
3 make && make install
```

Testing

To test your installation run `make test`.

Where To Start

Please visit the FAST wiki at https://github.com/paul-g/maxcc/wiki to get started.

[1] Please check the documentation of these tools for other dependencies.

Appendix B

Original RTM Kernel

The following listing shows the original compute kernel for the Reverse Time Migration application implemented with MaxJ using MaxCompiler 2012.1. Some typical Java constructs such as package definitions and imports are omitted.

```
1  public class RTM extends Kernel {
2    int Par=1, Mul=1, Sub=0;
3
4    public KArrayType<HWVar> burst_in=
5      new KArrayType<HWVar>(hwFloat(8, 24),  Par);
6
7    public KArrayType<HWVar> burst_out=
8      new KArrayType<HWVar>(hwFloat(8, 24),  Par);
9
10   public RTM(KernelParameters parameters) {
11     super(parameters);
12     HWFloat real    = hwFloat(8,24);
13     HWFix   fix_4_24= hwFix(4,24,HWFix.SignMode.TWOSCOMPLEMENT);
14
15     OffsetExpr nx = stream.makeOffsetParam("nx",  24/Par, 48/Par);
16     OffsetExpr nxy = stream.makeOffsetParam("nxy",32* nx, 32 * nx);
17
18     // Application Specific constants omitted
19     (...)
20     HWVar bc    = constant.var(-0.0005);
21
22     HWVar n1    = io.scalarInput("n1",    hwUInt(32));
23     HWVar n2    = io.scalarInput("n2",    hwUInt(32));
24     HWVar n3    = io.scalarInput("n3",    hwUInt(32));
25     HWVar ORDER = io.scalarInput("ORDER", hwUInt(32));
26     HWVar SPONGE= io.scalarInput("SPONGE",hwUInt(32));
27
28     CounterChain chain = control.count.makeCounterChain();
29     HWVar i4 = chain.addCounter(1000,1).cast(hwUInt(32));//iteration
30     HWVar i3 = chain.addCounter(n3,  1).cast(hwUInt(32));//outest loop
```

```
31      HWVar i2 = chain.addCounter(n2,  1).cast(hwUInt(32));
32      HWVar i1 = chain.addCounter(n1,Par).cast(hwUInt(32));//innest loop
33
34      HWVar up[] = new HWVar[Par];
35      for (int i=0; i <Par; i++)
36      up[i] = i3>=ORDER & i3<n3-ORDER  & i2>=ORDER & i2<n2-ORDER  & i1>=
            ORDER-i  & i1<n1-ORDER-i;
37
38      HWVar output_ring = n3  > i3  & i3 >= n3 - ORDER;
39
40      HWVar input_ring = ORDER > i3;
41
42      HWVar output_enable = (n3 - ORDER > i3 & i3 >= ORDER);
43
44      // input
45      KArray<HWVar> burst_p     =io.input("burst_p",       burst_in);
46      KArray<HWVar> burst_pp    =io.input("burst_pp",      burst_in);
47      KArray<HWVar> burst_dvv   =io.input("burst_dvv",     burst_in);
48      KArray<HWVar> burst_source=io.input("burst_source", burst_in);
49
50      HWVar p[]        =new HWVar[Par];
51      HWVar pp_i[]     =new HWVar[Par];
52      HWVar dvv[]      =new HWVar[Par];
53      HWVar source[]   =new HWVar[Par];
54
55      HWVar image[][] =new HWVar[Par][Mul];
56
57      for (int i=0; i <Par; i++)
58      {
59        p[i]=        burst_p[i].cast(real);
60        pp_i[i]=     burst_pp[i].cast(real);
61        dvv[i]=      burst_dvv[i].cast(real);
62        source[i]=   burst_source[i].cast(real);
63      }
64
65      HWVar cur[][][][]     = new HWVar[Mul][11+Par+1][11][11];
66      HWVar inter[][][]     = new HWVar[Par][Mul];
67      HWVar result[][]      = new HWVar[Par][Mul];
68
69
70      optimization.pushDSPFactor(1);
71      //Cache
72      for (int i=0; i <Par; i++)
73      {
74        int k = -6/Par;
75        for (int x=-6; x<=6; x+=Par)
76        {
77          for (int y=-5; y<=5; y++)
78            for (int z=-5; z<=5; z++)
79              cur[0][x+6+i][y+5][z+5] = stream.offset(p[i], z*nxy+y*nx+k);
80          k++;
81        }
82      }
```

```
83      //Computation
84      for (int i=0; i <Par; i++)
85      {
86      //data-path(0,i)
87      result[i][0]=(
88                      cur[0][6+i][5][5] * 2.0 - pp_i[i] +dvv[i]*(
89                      cur[0][6+i][5][5] * c_0
90                    +(cur[0][5+i][5][5] + cur[0][7+i][5][5]) * c_1_0
91                    +(cur[0][4+i][5][5] + cur[0][8+i][5][5]) * c_1_1
92                    +(cur[0][3+i][5][5] + cur[0][9+i][5][5]) * c_1_2
93                    +(cur[0][2+i][5][5] + cur[0][10+i][5][5])* c_1_3
94                    +(cur[0][1+i][5][5] + cur[0][11+i][5][5])* c_1_4
95                    +(cur[0][6+i][4][5] + cur[0][6+i][6][5]) * c_2_0
96                    +(cur[0][6+i][3][5] + cur[0][6+i][7][5]) * c_2_1
97                    +(cur[0][6+i][2][5] + cur[0][6+i][8][5]) * c_2_2
98                    +(cur[0][6+i][1][5] + cur[0][6+i][9][5]) * c_2_3
99                    +(cur[0][6+i][0][5] + cur[0][6+i][10][5])* c_2_4
100                   +(cur[0][6+i][5][4] + cur[0][6+i][5][6]) * c_3_0
101                   +(cur[0][6+i][5][3] + cur[0][6+i][5][7]) * c_3_1
102                   +(cur[0][6+i][5][2] + cur[0][6+i][5][8]) * c_3_2
103                   +(cur[0][6+i][5][1] + cur[0][6+i][5][9]) * c_3_3
104                   +(cur[0][6+i][5][0] + cur[0][6+i][5][10])* c_3_4 ))
105                   + source[i];
106     inter[i][0]   =(up[i])? result[i][0] : pp_i[i];
107     }
108     //Multiple Time-Dimension
109     for (int j=1; j <Mul; j++)
110     {
111       //Cache
112       for (int i=0; i <Par; i++)
113       {
114         int k = -6/Par;
115         for (int x=-6; x<=6; x+=Par)
116         {
117           for (int y=-5; y<=5; y++)
118             for (int z=-5; z<=5; z++)
119               cur[j][x+6+i][y+5][z+5] = stream.offset(inter[i][j-1], z*
                      nxy+y*nx+k);
120           k++;
121         }
122       }
123
124       //Computation
125       for (int i=0; i <Par; i++)
126       {
127       //data-path(j,i)
128       result[i][j]=(
129                       cur[j][6+i][5][5] * 2.0 - cur[j-1][6+i][5][5] +dvv
                            [i]*(
130                       cur[j][6+i][5][5] * c_0
131                     +(cur[j][5+i][5][5] + cur[j][7+i][5][5]) * c_1_0
132                     +(cur[j][4+i][5][5] + cur[j][8+i][5][5]) * c_1_1
133                     +(cur[j][3+i][5][5] + cur[j][9+i][5][5]) * c_1_2
```

```
134                       +(cur[j][2+i][5][5] + cur[j][10+i][5][5]) * c_1_3
135                       +(cur[j][1+i][5][5] + cur[j][11+i][5][5])* c_1_4
136                       +(cur[j][6+i][4][5] + cur[j][6+i][6][5]) * c_2_0
137                       +(cur[j][6+i][3][5] + cur[j][6+i][7][5]) * c_2_1
138                       +(cur[j][6+i][2][5] + cur[j][6+i][8][5]) * c_2_2
139                       +(cur[j][6+i][1][5] + cur[j][6+i][9][5]) * c_2_3
140                       +(cur[j][6+i][0][5] + cur[j][6+i][10][5])* c_2_4
141                       +(cur[j][6+i][5][4] + cur[j][6+i][5][6]) * c_3_0
142                       +(cur[j][6+i][5][3] + cur[j][6+i][5][7]) * c_3_1
143                       +(cur[j][6+i][5][2] + cur[j][6+i][5][8]) * c_3_2
144                       +(cur[j][6+i][5][1] + cur[j][6+i][5][9]) * c_3_3
145                       +(cur[j][6+i][5][0] + cur[j][6+i][5][10])* c_3_4 ))
146                       + source[i];
147         inter[i][j]   =(up[i])? result[i][j] : cur[j-1][6+i][5][5];
148         }
149       }
150
151       //setup configuration
152       optimization.popDSPFactor();
153
154       // control counter
155       KArray<HWVar> output_p  = burst_out.newInstance(this);
156       KArray<HWVar> output_pp = burst_out.newInstance(this);
157
158       for (int i=0; i <Par; i++)
159       {
160         output_p[i] <==       inter[i][Mul-1].cast(hwFloat(8,24));
161         output_pp[i] <== cur[Mul-1][6+i][5][5].cast(hwFloat(8,24));
162       }
163
164       io.output("ker_p",     output_p,  burst_out);
165       io.output("output_pp", output_pp, burst_out);
166
167     }
168   }
```

Appendix C

Original Memory Read Kernel

The following listing shows the original memory read kernel for the Reverse Time Migration application implemented with MaxJ using MaxCompiler 2012.1. This kernel generates the memory commands required to fetch data from DRAM. Some typical Java constructs such as package definitions and imports are omitted.

```
1  public class Cmdread extends Kernel {
2      public Cmdread(..) {
3        int    Burst_inc      =1;
4        HWVar iniBursts      =io.scalarInput("iniBursts",hwUInt(32));
5        HWVar totalBursts    =io.scalarInput("totalBursts",hwUInt(32));
6        HWVar wordsPerBurst =io.scalarInput("wordsPerBurst",hwUInt(32));
7        HWVar Enable         =io.scalarInput("Enable",hwUInt(1));
8
9        //the address counters
10       Count.Params param0 = control.count.makeParams(32)
11             .withEnable(Enable).withMax(wordsPerBurst).withInc(1);
12       Counter counter0     = control.count.makeCounter(param0);
13       HWVar  wordCount     = counter0.getCount();
14       Count.Params param1 = control.count.makeParams(32)
15             .withEnable(counter0.getWrap()).withMax(totalBursts).withInc(
                   Burst_inc);
16       Counter counter1     = control.count.makeCounter(param1);
17       HWVar burstCount     = counter1.getCount();
18       HWVar Control        = wordCount.eq(0) & Enable;
19       DRAMCommandStream.makeKernelOutput("dram_read",
20            Control,                              // control
21            burstCount+iniBursts,                 // address
22            constant.var(hwUInt(8),Burst_inc),// size
23            constant.var(hwUInt(6),  1),          // inc
24            constant.var(hwUInt(4),  0),          // stream
25            constant.var(false));
26
27     }}
```

Appendix D

FAST Dataflow Kernels

D.1 Numerical Differentiation

The FAST dataflow kernel that implements the differentiation operation is shown in the Listing below. It uses no API (non-user defined) function calls and a total of 20 lines of code.

```
1  #include ''fastc/fast.h''
2
3  const int Par;
4
5  int stream_diff(float *value[Par], int offset, int pipe) {
6    int cycle_offset = (pipe + offset) / Par;
7    int pipe_offset  = (pipe + offset) % Par;
8    int cycle_offset_neg = (Par - 1 - pipe + offset) / Par;
9    int pipe_offset_neg  = Par - 1 - (Par - 1 - pipe + offset) % Par;
10   return value[pipe_offset][cycle_offset] -
11          value[pipe_offset_neg][cycle_offset_neg];
12 }
13
14 int diff(float *value[Par], float h, int pipe) {
15   return (-2 * stream_diff(value, 2, pipe)
16           -1 * stream_diff(value, 1, pipe)) / (10 * h);
17 }
18
19 void kernel_SGDiff( float* value[Par], int width, int size, double step)
       {
20   int cycle = Par * CYCLE_COUNT + pipe;
21   bool compute = (cycle >= width) & (cycle < size - width);
22
23   for (int pipe = 0; pipe < Par; pipe++) {
24     result[pipe] = compute ? diff(value, h, pipe) : value[pipe]
25   }
```

```
}
```

D.2 Add Prediction

```
#include "../../../include/maxcc.h"

float PDF(float z) {
  float root2pi = 2.50662827463100050242;
  return exp(-z * z / 2) / root2pi;
}

float CDF(float z) {

 // constants p0..p6 and q0..q6 are omitted
 (...)

  float cutoff = 7.071;

  float root2pi = 2.50662827463100050242;

  float zabs = sqrt(z);
  float expntl = exp(-0.5 * zabs * zabs);
  float pdf = expntl / root2pi;

  bool c1 = z > 37.0;
  bool c2 = z < -37.0;
  bool c3 = zabs < cutoff;

  float pA = expntl *
    ((((((p6 * zabs + p5) * zabs + p4) * zabs + p3)* zabs + p2) * zabs +
        p1) * zabs + p0) /
    (((((((q7 * zabs + q6) * zabs + q5)*zabs + q4) * zabs + q3) * zabs +
        q2)*zabs + q1 * zabs) + q0 * zabs);

  float pB = pdf / (zabs + 1.0/(zabs + 2.0/(zabs + 3.0/(zabs + 4.0/(zabs
      + 0.65)))));

  float pX = c3 == 0 ? pB : pA;
  float p = (z < 0.0) ? pX : 1 - pX;
  return c1 == 0? (c2 == 0 ? p : 0.0) : 1.0;
}

float V(float t) {
   float cdf = CDF(t);
  return (cdf == 0) ? 0 : PDF(t) / cdf;
}

float W(float t) {
```

```
42    float v = V(t);
43    return v * (v + t);
44  }
45
46  const int N = 10;
47  #pragma fast var:y ioType:float(8, 24) computeType:float(8,12) func:
      kernel_Adp
48  void kernel_Adp(
49                  float y, float beta,
50                  float* prior_m[10],
51                  float* prior_v[10],
52                  float* post_m[10],
53                  float* post_s[10])
54  {
55    float m, s;
56    m = 0;
57    s = 0;
58    for (int i = 0; i < N ; i++) {
59      m = m + prior_m[i][0];
60      s = s + prior_v[i][0];
61    }
62
63    float S = sqrt(beta * beta + s);
64    float t = (y * m) / S;
65
66    for (int i = 0; i < N; i++) {
67      float pr_m = prior_m[i][0];
68      float pr_v = prior_v[i][0];
69      float ps_m = pr_m + y * (pr_v / S) * V(t);
70      float a = abs(pr_v * 1 - ( (pr_v / (S * S)) * W(t)));
71      float sq = sqrt(a);
72      float ps_s = pr_v + sq;
73      post_m[i][0] = ps_m;
74      post_s[i][0] = ps_s;
75    }
76
77  }
```

Appendix E

Experimental Data

This chapter contains experimental data for the application benchmarks that were omitted from the main report body for the sake of brevity.

Design space exploration results for the Reverse Time Migration application are shown in Table E.1

Experimental speedup results for our implementation of the bitonic sorting network design are listed in Table E.2.

Aspects	DSP Balance	Par	Mem Freq	Width	FPGA Time (Ms)	Total Time (s)	Total Power (Watt)	Dyn. Power (Watt)	LUT (%)	FF(%)	BRAM (%)	DSP (%)	Build Time
OptimizeDSPUsage, ExploreParallelism	Bal.	1	303	8_24	492	3.347	117	30	19.89	14.13	23.31	1.69	1h49
	None				492	3.453	117	30	22.62	15.56	23.31	0	2 h 9
	Full				492	3.103	117	30	17.68	12.79	23.31	5.8	2 h 2
	Bal.	2			247	2.917	121	34	25.44	25.44	24.53	3.37	2h30
	None				247	3.006	121	34	30.95	20.53	24.53	0	3h5
	Full				246	3.153	121	34	21.18	15	24.44	11.61	2h8
	Bal.	3			225	3.506	121	34	31.56	21.32	24.06	5.06	2h30
	None				224	3.124	121	34	39.16	25.61	24.06	0	3h20
	Full				226	3.123	125	38	25.06	17.3	23.97	17.41	2h18
	Bal.	6			226	3.125	125	38	52.61	30.52	20.12	17.41	3h17
	None				223	3.156	125	38	65.11	40.58	28.95	0	4h50
	Full				224	3.016	122	35	35.63	24.11	27.73	34.82	3h42
OptimizeWordWidth, Optimize-Frequency, ExploreParallelism	Full	6	303	8_22	224	2.98	122	35	45.02	31.17	27.73	15.18	3h13
				8_20	224	3.006	122	35	43.54	29.96	27.16	15.18	3h30
				8_18	224	3.076	122	35	40.81	28.68	26.88	15.18	3h13
				8_16	224	3.033	122	35	39.62	26.44	26.6	10.12	2h50
				8_24	82	1.723	122	35	34.13	24.09	27.73	34.82	4h17
		2	400	8_24	247	3.133	122	35	21.32	14.98	24.44	11.61	2h30
		3		8_24	164	2.964	122	35	25.43	17.28	23.97	17.41	2h35

Table E.1: Design space exploration results for the RTM benchmark application.

Network Size	log(n)	CPU Time	FPGA Time	Compute Speedup	FPGA Total Time	FPGA Total Speedup
128	14	0.156162	0.006564	23.79	1.006564	0.1551
	15	0.312428	0.013028	23.98	1.013028	0.3084
	16	0.624328	0.026263	23.77	1.026263	0.6084
	17	1.24981	0.051905	24.08	1.051905	**1.1881**
	18	2.49068	0.102083	24.4	1.102083	2.26
	19	4.98994	0.199928	24.96	1.199928	4.1585
	20	9.97521	0.400709	24.89	1.400709	7.1215
	21	19.9636	0.807231	24.73	1.807231	11.0465
	22	39.9202	1.614437	24.73	2.614437	15.2691
	23	79.8726	3.203924	24.93	4.203924	18.9995
64	16	0.271163	0.012983	20.89	1.012983	0.2677
	17	0.541016	0.025841	20.94	1.025841	0.5274
	18	1.08533	0.052019	20.86	1.052019	**1.0317**
	19	2.16697	0.10265	21.11	1.10265	1.9652
	20	4.33626	0.203208	21.34	1.203208	3.6039
	21	8.66081	0.406241	21.32	1.406241	6.1588
	22	17.3281	0.806145	21.5	1.806145	9.594
	23	34.6705	1.617021	21.44	2.617021	13.2481
32	18	0.458403	0.026085	17.57	1.026085	0.4467
	19	0.917055	0.051485	17.81	1.051485	0.8722
	20	1.83439	0.100844	18.19	1.100844	**1.6663**
	21	3.67453	0.203696	18.04	1.203696	3.0527
	22	7.35396	0.404244	18.19	1.404244	5.237
	23	14.6802	0.810293	18.12	1.810293	8.1093
16	19	0.375978	0.026123	14.39	1.026123	0.3664
	20	0.759506	0.051846	14.65	1.051846	0.7221
	21	1.49907	0.102618	14.61	1.102618	**1.3596**
	22	3.00802	0.201457	14.93	1.201457	2.5036
	23	5.99392	0.406334	14.75	1.406334	4.2621

Table E.2: Speedup results for the FPGA sorting network compared to the CPU only version. Points after which it becomes convenient to use FPGA acceleration are highlighted.