NTNU
Norwegian University of
Science and Technology

Accelerating Adaptive Mesh Refinement through Multiscale Dataflow Computing

Morten Johannes Barbala

Master of Science in Computer Science
Submission date: August 2018
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer Science

Project Description

In this project we want to test various dataflow kernels of HPC-related computations using a dataflow computing system that was donated to NTNU by Maxeler. The purpose is to explore the benefits of FPGA acceleration and see how the Maxeler system eases the typically hard and time-consuming FPGA development process. The main focus of the project is to accelerate miniAMR, a proxy application for adaptive mesh refinement developed by Mantevo Project.

Abstract

As we observe diminishing returns for multi-core CPUs, especially when considering power budgets, FPGAs are becoming increasingly important in the HPC world. To push the limits of performance and energy efficiency, more general-purpose hardware, i.e. multi-core and GPU systems are often not sufficient, and we need to create specialised hardware systems. FPGAs are specialised hardware devices that allow us to create systems that are optimized for a given application. However, development and integration is generally difficult and time-consuming.

In this thesis we explore harnessing the power of FPGA acceleration through Maxeler's FPGA-based Multiscale Dataflow Computing system by accelerating miniAMR, a proxy application for adaptive mesh refinement developed by Mantevo project. Proxy applications are minimal applications, which mimic the performance characteristics of full applications and are meant for testing and benchmarking. The applications are easier to work with than full applications and as they are meant to test both hardware and software, they contain a lot of options and runtime arguments, which makes FPGA acceleration challenging.

Using the Maxeler system, we create arithmetic kernels for the core 7-point and 27-point stencil computations of miniAMR. By rearranging the data used, properly managing memory, and moving the core 3D stencil calculations onto dataflow engines, we achieve a maximum speedup of 2.52 while maintaining the functionality of miniAMR. Because of the flexibility of the Maxeler system and since the application mimics characteristics of full applications, our kernels can potentially be used to accelerate full adaptive mesh refinement or other stencil-driven applications with minimal effort.

Sammendrag

I dag ser vi mer og mer begrenset ytelse fra flerkjerneprosessorer, spesielt hvis man tar strømbruk og kjøling med i betraktningen, noe som betyr at FPGAer blir stadig mer viktig for tungregning. For å teste grensene til ytelse og energieffektivitet er mer generell maskinvare ikke alltid nok og vi må lage spesialiserte systemer. FPGAer er spesialiserte enheter som kan brukes til å lage systemer som er optimalisert for en gitt applikasjon, men utvikling og integrering er typisk vanskelig og tidkrevende.

I dette prosjektet utforsker hvordan vi kan utnytte FPGA-akselerering gjennom Maxeler sitt "Multiscale Dataflow Computing" system som baserer seg på FPGAer ved å akselerere miniAMR, en proxyapplikasjon for tilpasset raffinering av nett utviklet av Mantevo prosjektet. Proxyapplikasjoner er minimale applikasjoner som etterligner egenskaper til fullstendige applikasjoner og brukes til testing og benchmarking. Applikasjonene er lettere å jobbe med enn fullstendige applikasjoner og siden de er ment for å teste både maskinvare og programvare så har de mange alternativer og kjøretidsargumenter som gjør FPGA-akselerering utfordrende.

Ved hjelp av Maxelersystemet har vi laget aritmetiske kjerner for 7-punkts- og 27-punktsstensiler som ligger i kjernen av beregningen for miniAMR. Ved å rearrangere dataen som bruker, håndtere minnet på en skikkelig måte, og flytte de aritmetiske kjernene for stensilberegningene over på dataflytmaskiner så oppnår vi en ytelsesforbedring på 2.52 mens vi beholder all funksjonalitet i miniAMR. På grunn av fleksibiliteten til Maxelersystemet og siden applikasjonen etterligner egenskapene til en fullverdig applikasjon så kan de aritmetiske kjernene våre potensielt bli brukt til å akselerere fullverdige applikasjoner for tilpasset raffinering av nett eller andre stensilapplikasjoner med minimal innsats.

Acknowledgements

First and foremost I have to thank all my fellow members of the HPC-Lab at NTNU for support, collaboration and friendship. You have all made the last year truly unforgettable. I would also like to thank Dr. Anne C. Elster for being my supervisor and managing the HPC-Lab to get us all the resources we need (and some we do not need, but greatly appreciate). Thanks to NTNU for their support of the HPC-Lab.

Many thanks to Maxeler for donating the dataflow system to the HPC-lab, originally as part of the collaborations for the EU H2020 Cloud Lightning project. Thanks to Maxeler and Dr. Anne C. Elster for giving me the opportunity to travel for a course on dataflow computing at Maxeler's headquarters in London as in introduction to this Master's project. Special thanks go to Tobias Becker at Maxeler for being a valuable resource and providing support and guidance.

Finally, I would like to thank the Hartree Centre, and especially Michael Bane, for giving me access to their Maxeler system for development, testing and simulation.

Table of Contents

List of Figures

List of Listings

List of Abbreviations

API application programming interface

ASIC Application-specific integrated circuit

CLB configurable logic block

CPU central processing unit

DFE dataflow engine

FMem fast memory

FPGA field-programmable gate array

GPU graphics processing unit

HDL hardware description language

HPC high-performance computing

I/O input/output

LMem large memory

LUT lookup table

NTNU Norwegian University of Science and Technology

PCIe Peripheral Component Interconnect Express

SLiC simple live CPU interface

VHDL VHSIC hardware description language

Chapter 1

Introduction

FPGAs are interesting pieces of hardware for modern high-performance computing. They are flexible and allow us to create ideal machines for the type of computation we need for a given application. Today, acceleration is done either in the form of parallel programming using general-purpose hardware, e.g. multi-core CPUs and GPUs, or by using specialised hardware, e.g. FPGAs and ASICs. As we observe diminishing returns from multi-core systems, FPGAs are becoming increasingly important alongside GPUs [6].

FPGAs help bridge the gap between general purpose and specialised hardware by being programmable and configurable. However, FPGA acceleration is difficult and time-consuming. This type of computation is less mature than parallel computation using multi-cores and GPUs and the process of converting standard high-level code to efficient parallel code is better understood than conversion to efficient hardware designs suited for FPGAs. FPGAs also run with far lower operating frequencies, and to achieve better performance we need to use massive parallelism and deep pipelines [1, 11].

Maxeler's Multiscale Dataflow Computing makes FPGA acceleration more accessible by providing a system where dataflow is emulated on FPGA-based dataflow engines and development takes a high-level approach. Dataflow is the concept of viewing programs, or computation, as directed graphs with clear data dependencies and deterministic execution, which makes it well suited to represent massive parallelism and deep pipelines [5]. Dataflow engines try to overcome some of the challenges associated with FPGAs acceler-

ation by operating with arithmetic kernels developed in Java and integration with applications through basic function calls using a SLiC interface [9].

To test the Maxeler system, we seek to accelerate miniAMR, a proxy application developed by Mantevo project. Proxy applications are minimal implementations that mimic the performance characteristics of full applications and are meant to benchmark and test both software and hardware [7]. The applications typically contain a lot of runtime variability, which makes them challenging to accelerate.

Chapter 2 covers necessary background theory: How FPGAs work, FPGA acceleration, dataflow, and the Maxeler system as well as how to program DFEs. Chapter 3 covers the process of accelerating miniAMR and Chapter 4 shows our results. Finally, Chapter 5 concludes and lists future work.

Chapter 2

Background

This chapter covers the necessary background theory for this project. We cover FPGAs, dataflow, and both the hardware aspects and programming of the Maxeler system. Section 2.1 describes how FPGAs work, the development process and the potential for increased performance and HPC. Section 2.2 describes dataflow and the most common execution model. Section 2.3 to Section 2.5 covers Maxeler's Multiscale Dataflow Computing system and dataflow computers, and finally Section 2.6 describes dataflow programming using the Maxeler system.

2.1 Field-Programmable Gate Array

Field-programmable gate array (FPGA) is the name of devices consisting of a matrix of *configurable logic blocks* (CLBs) and programmable interconnects, or switches. This structure is shown in Figure 2.1. The devices are called field-programmable because configuration is possible "in the field", i.e. after manufacturing. FPGAs are integrated circuits which can be reprogrammed, or reconfigured, with the functionality required for a given application. The logic blocks are configured to perform specific operations and switches are configured to provide an interconnection between the logic blocks [2, 3].

CLBs are created by combining a *lookup table* (LUT) with a flip-flop (see Figure 2.2). The LUTs contains a block memory that is written to in order

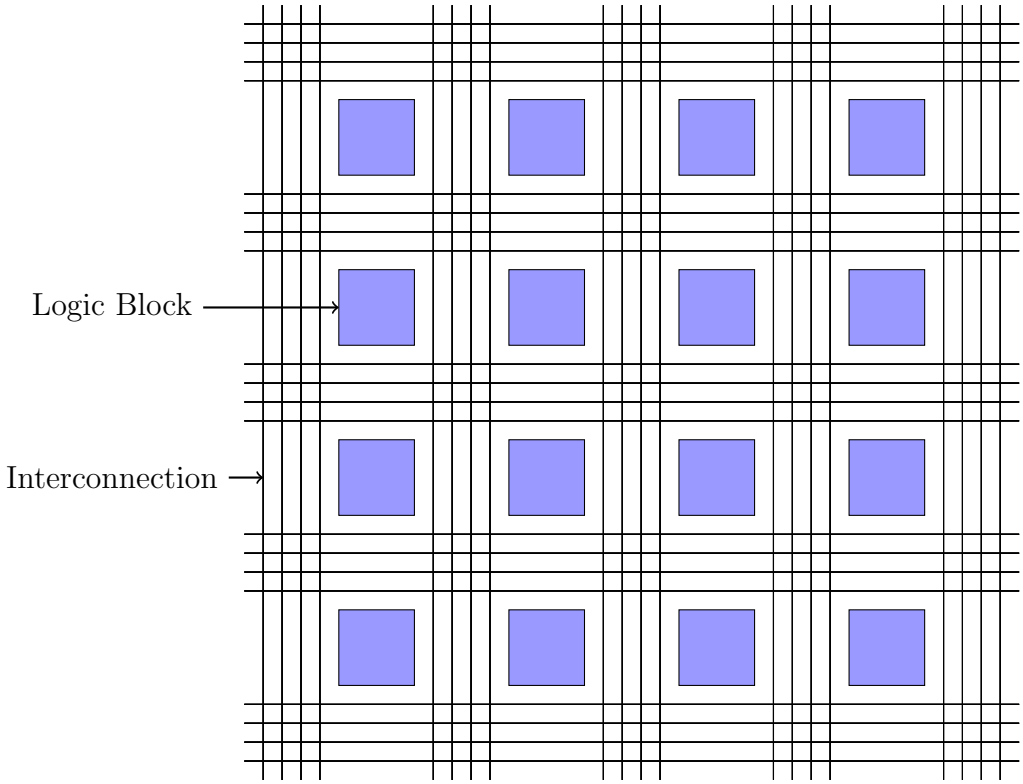


Figure 2.1: Internal structure of FPGA chips

to implement a certain combinational function and inputs to the logic blocks are used as addresses by the configured LUT to determine which value to look up and produce as output. This output can either be sent out directly for a fully combinational implementation or go through the flip-flop to make the block sequential. Sequential behaviour is needed to be able to store the state of the computation between clock cycles and create essential components like registers and finite state machines [2, 3].

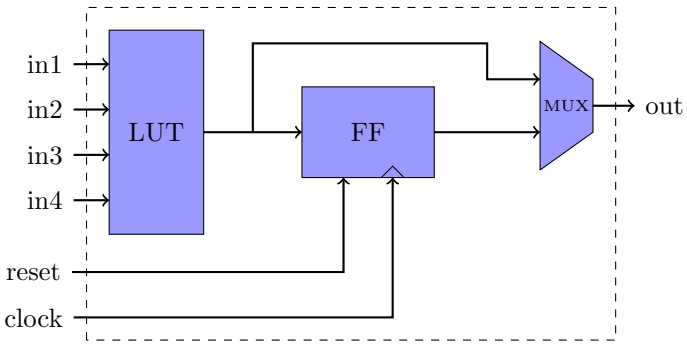


Figure 2.2: Key components of CLBs

2.1.1 FPGA Acceleration

FPGAs were originally developed in the 1980s to enable quick and easy prototyping. The programmability made FPGA devices suitable to emulate hardware circuits, which meant digital designs could be prototyped, tested and debugged before actual circuits were manufactured. The devices were also suitable as teaching tools, but by the early 2000s, the use of FPGAs had spread to most areas of computing and there was also adoption in certain specific areas when it came to HPC, namely cryptography and specialized arithmetic. The strength of FPGAs lies in possibility for extreme parallelism and optimal hardware solution. Typical overhead instructions for software solutions like loops, branches and array indexing are also completely avoided and every operation can produce a payload [11].

The road towards widespread adoption for HPC started fully between 2003 and 2006 when it was observed that Dennard scaling was breaking down. Moore's law was still in effect, albeit slowing down, and it was no longer economically viable to support higher frequencies, mainly because of the

increasing power density. Consequently, the focus shifted from single-core performance to parallelism with multi-core processors and GPUs. It eventually became clear that FPGAs had potential in the HPC world and today FPGAs are used to create, or emulate, "ideal machines" for a given application. The configurability enables developers to implement optimal hardware solutions, and create new forms of computations that are not available in CPUs or GPUs, e.g. dataflow computing, which is described in Section 2.2. The three main reasons for the potential of FPGAs are [4, 11]:

1. The scalability problem with multi-cores: It is difficult to scale up, both for frequency and number of cores, especially while limited by power targets and cooling capabilities.

2. Unlike power density and frequency scaling, Moore's law on transistor density scaling still applies, which enable huge FPGA configurations. FPGAs have the potential for implementing large, complex computation engines, and since they inherently operate at lower frequencies, the power density problem has not been hit yet.

3. FPGAs have extreme flexibility and can implement practically any hardware configuration needed for a given application. Most forms of parallelism and computation can be used, and optimal configurations can be created for applications of any scope.

2.1.2 FPGA Development

The FPGA development flow consists of four major steps: Creating the design, creating test benches for simulations, synthesis and implementation, and finally programming file generation. Design of the system results in *hardware description language* (HDL) files and can either be done directly in traditional HDLs like VHDL or Verilog, or with the help of various tools that use high-level techniques to generate the HDL [1, 11]. Creating test benches is done to perform simulations and verify that the design is working as intended before it is implemented on the FPGA. The synthesis is what converts the design to a basic hardware description in the form of gate-level components, which can be implemented on the FPGA. The implementation consists of smaller processes that generate the physical placement and the FPGA chip, i.e. locations of the logic block and routes between them. The final step

is generating a programming file, or bitstream, which can be downloaded, a.k.a. flashed, onto the FPGA, after which it will function as a digital circuit performing the functionality specified by the design [2].

Using FPGAs for HPC can result in significant performance improvements, as well as increased energy efficiency. Extreme parallelism is possible, especially for computations using reduced precision, and typical overhead instructions like array indexing and loop computations are avoided since the control flow is configured directly in the logic, trading space for performance. However, FPGAs, have some big challenges including low operating frequency and programmability. FPGAs typically run at a frequency ten times lower than high-end processors. Furthermore, the process of transforming existing code, to run on multi-core processors or GPUs is more mature and better understood than for FPGAs. Achieving significant speedup is usually possible, but entails making trade-offs between development effort and flexibility, portability, maintainability etc [4].

2.2 Dataflow

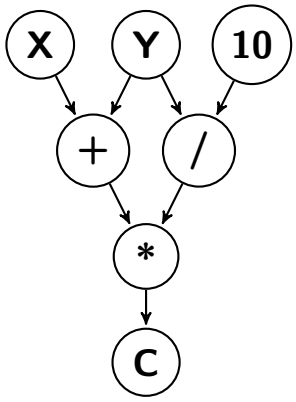
Dataflow is the concept of viewing programs as directed graphs. The systems for representing the programs can use cyclic graphs for full functionality and support, e.g. loops, or acyclic graphs for easier optimization and generation of efficient hardware designs, for example for FPGAs [1]. The nodes in the graphs represent simple arithmetic or logical operations, i.e. basic instructions, while the arcs represent data dependencies between the instructions. The name "dataflow" comes from the concept of data flowing between the instructions along the arcs. A simple program and its corresponding dataflow graph is shown in Figure 2.3 [5].

2.2.1 Execution Model

The most common model for dataflow execution is the token-based dataflow model in which data flows as tokens along the arcs in theoretically endless FIFO queues. Arcs flowing into nodes are called input arcs and arcs flowing out are called output arcs, and the set of input arcs that are needed for the node to perform its operation is called the *firing set*. When all arcs in

$$A = X + Y$$
$$B = Y/10$$
$$C = A * B$$

(a) A basic arithmetic program



(b) A simple dataflow graph

Figure 2.3: A basic arithmetic program in a traditional representation (a) and dataflow (b), adapted from Johnston, Hanna, and Millar [5]

the firing set has data, the node is what we call *fireable*. After a node has become fireable, it is activated, i.e. performs its operation, at some undefined time. Execution consists of removing data tokens from arcs in the firing set, performing the operation and creating new data tokens to place on output arcs before waiting to become fireable again. This results in instructions being executed, or scheduled for execution in practice, as soon as operands are available, which is fundamentally different from the classic von Neumann model where instructions are scheduled accordingto the program counter. [5]

The main advantage of the dataflow model is that more than one instruction can be executed at once. There are widely used techniques to add this capability to conventional processors, e.g. multi-core and multiple issue, but the dataflow model has an inherent potential for extreme parallelism at the instruction level. Programs in the pure dataflow model are also determinate. This means that for a given set of inputs, execution will always produce the same outputs. This is because the operations of the nodes are functional, i.e. data is never modified, there is a locality effect and there are no side effects. Languages for dataflow programming also need this functional behaviour to be efficient, and most dataflow programming languages will therefore be largely functional languages. However, some imperative constructs, or at least imperative syntax, is needed to implement loops, so

dataflow languages can also be created by restricting imperative languages. The most common approach is to utilize high-level non-functional techniques and meta-programming through abstractions, which are compiled down to code suited for dataflow or create hardware configurations which emulate dataflow. One such system, Maxeler's Multiscale Dataflow Computing, is described in Section 2.3 [5, 11].

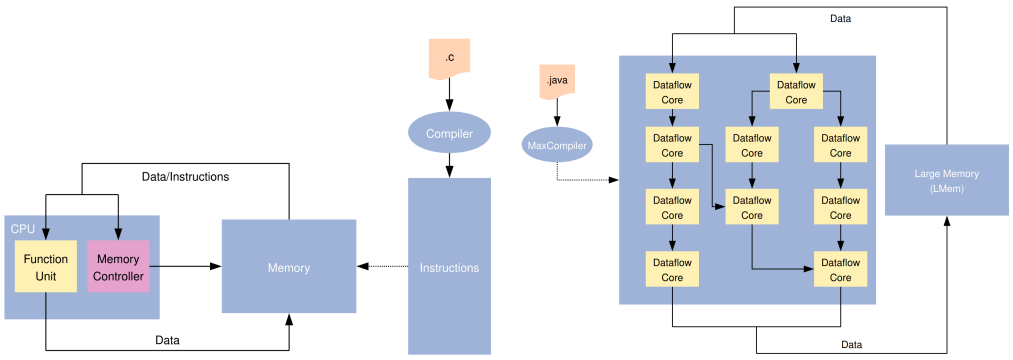
2.3 Multiscale Dataflow Computing

The Maxeler model is an evolution of the dataflow architectures from the 1970s and 1980s. Multiscale dataflow computers combine a fast standard processor with DFEs, which are emulated on FPGAs. Traditional CPUs are inherently sequential and programs typically contain a critical loop where memory is read and data is moved around, which limits the maximum speed of computation. Dataflow engines does away with this problem by operating on streams of data that flow between functional units called *dataflow cores* on the chips. Tiny on-chip memories are spaced out to create a distributed register file, which can have as many access ports as needed to ensure smooth flow of data [9].

Dataflow is applied at various levels of abstraction: System level, architecture level, arithmetic level and bit level. On a system level, multiple dataflow engines are connected within a single computer system and multiple systems can be connected to form clusters or super computers. On an architecture level, memory access and arithmetic or logical operations are decoupled, i.e. I/O orchestration is separated from computation. On arithmetic and bit level, representations of data can be optimized, e.g. using reduced precision, to balance computation and communication [9].

Each functional unit only performs one operation, arithmetic or logic, and are simple enough that one DFE can accommodate thousands of dataflow cores, all executing concurrently. This approach is said to be "computing in space" as opposed to the "computing in time" of traditional control flow cores, because the complete computation for a given application is laid out spatially on the chip rather than taking place at different time points on a smaller number of functional units (see Figure 2.4). Data dependencies are therefore resolved at compile time, which simplifies the implementation of

extreme parallelism and deep pipelines. The whole dataflow engine can be pipelined with anywhere from 1000 to 10000 stages [10].



(a) Control flow system (b) Dataflow system

Figure 2.4: Control flow (a) vs. dataflow (b), from *Multiscale Dataflow Programming, version 2016.1.1* [9]

The Maxeler dataflow computation model does not explicitly operate with instructions. The structure of the dataflow engines represent the computation and the instructions are replaced by execution units, i.e. the dataflow cores. This means there is no overhead from decoding instructions, and techniques such as branch prediction and out-of-order scheduling are not needed, and general purpose caches are unnecessary because data is constantly available on the chip where it is needed. In other words, the majority of the chip area can be utilized for computation, but this does require deep pipelining and general parallelism for efficiency, and the dataflow engines are less suited for single operations, especially on small amounts of data. Computer systems also need control flow for various other tasks, and this is why Maxeler combines their dataflow engines with a conventional CPU as described in Section 2.4 [10].

2.4 Dataflow Computers

A dataflow computer is typically a heterogeneous system that consists of multiple dataflow engines and a standard CPU. The CPU is used for administrative, system level tasks like storage, networking and process/data

management in addition to smaller workloads, which are better suited for control flow. The two types of devices serve different purposes, and there are multiple schemes for combining them in one system. One simple scheme is taking a master-slave approach with the CPU operating as the master. The DFEs will then take commands and data from the CPU in a passive manner before performing the computation. The simplicity of this scheme is the main advantage, but the CPU and DFEs might need to operate equally in the system, i.e. with the same access to system resources and the same priority. The DFEs can then instruct the CPU for operations, which will lead to better performance for certain applications. The main challenge for this approach is designing the interface between DFEs and CPUs [10].

2.4.1 Memory

Memory management and data handling are generally the most important aspects of HPC. When memory access and data orchestration is a significant part of the given application, optimization on processor architecture or arithmetic kernels will quickly hit diminishing returns. One approach for memory access is to implement a combined, unified address space for CPU and DFEs. This is the most flexible approach, but introduces problems with data coherence between devices and contention on data buses, which limit the achievable performance. The most used approach is therefore to provide the DFEs with multiple memory banks. This gives greater bandwidth and divides the memory hierarchy to potentially give maximum performance, but also results in more complexity and increased development effort. To make programming easier, software abstractions and libraries are typically provided for memory management. The Maxeler DFEs have two types of memories: *fast memory* (FMem), which lies on-chip, and *large memory* (LMem), which lies off-chip. FMem can store a couple of megabytes of data and is capable of access bandwidth reaching terabytes per second, while LMem can store many gigabytes of data, but is significantly slower [10, 9].

2.4.2 Coupling

Another important aspect of dataflow computers is how the CPU and DFEs are connected, or coupled. The tightest coupling (see Figure 2.5a) is achieved

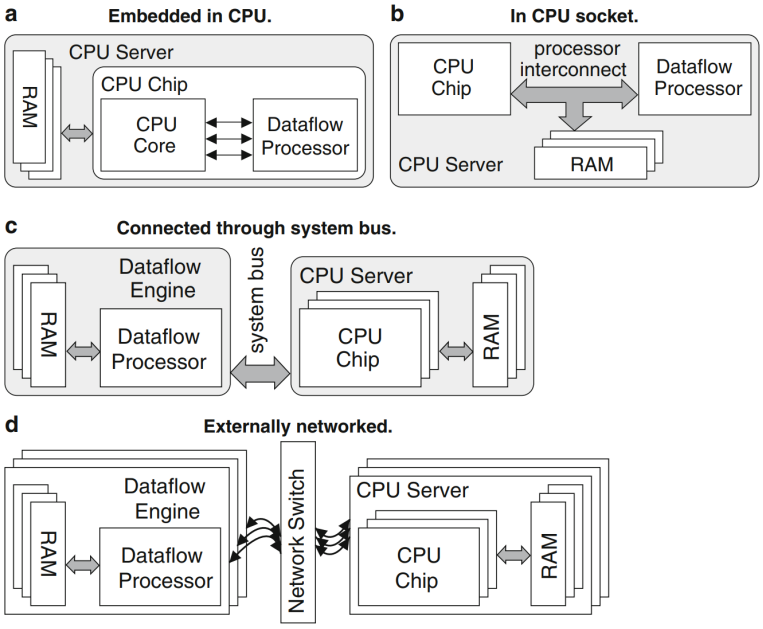


Figure 2.5: Interconnection approaches for dataflow computers, from Pell et al. [10]

by placing all devices on the same silicon die and having dedicated low latency interfaces. With this approach the DFEs are typically activated through special instructions on the CPU. The approach gives the most efficient communication, but it is only viable for small operations and requires modification of the CPU architecture. Another approach for tight coupling (see Figure 2.5b) is to provide sockets for the DFEs similarly to CPUs multi-CPUs systems. The DFEs operate as standard CPUs and communication is done through conventional processor interconnects. The major downside is that the DFEs must support all features required to interface with the socket, which takes resources away from computation, and this approach is also limited in size and scale. A third approach (see Figure 2.5c) is to connect the DFEs using PCI Express or other system buses. This way the interface can be implemented either in the processor similarly to using dedicated sockets, or in chip-external components on the DFEs. This results in overhead both for communication latency and silicon area, but modern system bus standards have enough bandwidth to accommodate a few DFEs per CPU. Finally, the last approach (see Figure 2.5d) is a loosely coupled one that is related to

cloud computing. Using clusters of CPUs and DFEs, communication is done through the network interconnects, e.g. Ethernet or InfiniBand. This approach trades latency for greatly increased flexibility [10].

2.5 Real World Implementations

Maxeler offers three real world implementations of dataflow computing systems: MPC-C, MPC-X and MPC-N. MPC-C uses the system bus approach shown in Figure 2.5c, while MPC-X uses the cluster approach. MPC-C systems (see Figure 2.6) combine x86 CPUs connected with DFEs through PCI Express, and the DFE are connected with a dedicated low-latency and high-bandwidth interconnect called *MaxRing*. MPC-N systems are similar to MPC-C systems except they have fewer DFEs, no MaxRing connection and an emphasis on networking and throughput rather than computing. Both systems are used to implement dataflow computation with a fixed combination of CPUs and DFEs. The MPC-X systems also uses MaxRing interconnect between DFEs, but are stand-alone dataflow nodes, which need to be connected with other nodes in clusters or supercomputers. More heterogeneous systems with multiple computation technologies and a need for varying the number of DFEs require the MPC-X series [10, 9].

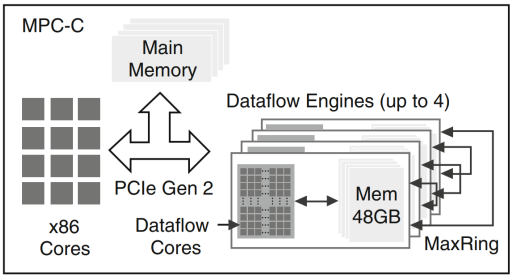


Figure 2.6: MPC-C Series Architecture, from Pell et al. [10]

2.6 Maxeler Dataflow Programming

The Maxeler programming model for dataflow, which is shown in Figure 2.7, involves developing three separate parts of the application: CPU code, dataflow

kernels, and a manager. The CPU code is written in any supported high-level language, typically C or FORTRAN, while dataflow kernels and the manager are written in MaxJ, a meta language based on Java. Maxeler has created a language extension for Java to generate the hardware configuration for the dataflow engines. The CPU code runs as a normal CPU application and controls the system. The dataflow kernels, or arithmetic kernels, are datapaths which perform the computation needed for the application. The manager controls the flow of data as streams from CPU, between kernels, and between on-chip and off-chip memory. The key part of this model is that computation and communication is split into kernels and the manager, respectively, which enables development of deeply pipelined kernels without typical problems associated with parallelism, e.g. synchronization. To achieve maximum performance, one must take advantage of the potential for deep pipelining and parallelism both between kernels and within kernels [10].

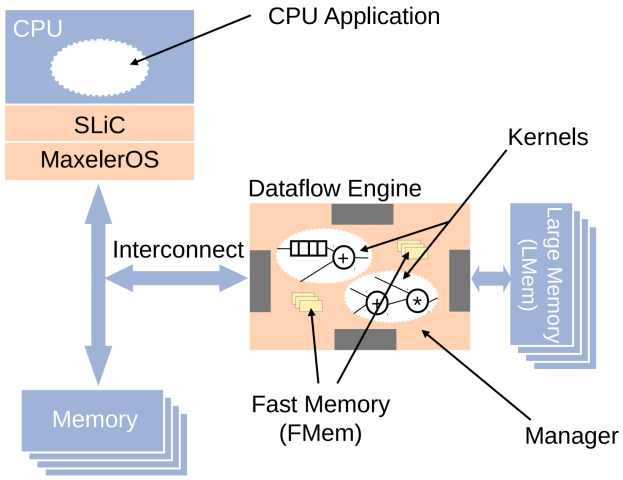


Figure 2.7: Maxeler dataflow programming model, from *Multiscale Dataflow Programming, version 2016.1.1* [9]

2.6.1 Dataflow Application

Complete application with dataflow acceleration will typically consist of mainly CPU code with some small parts, i.e. arithmetic kernels, running

on dataflow engines. When executing the Java code, the kernels are translated into graphs of pipelined arithmetic units. If there are no loops in the design, the data simply flows through the graph, i.e. between inputs and outputs, and as long as there is more data than pipeline stages in the implementation, the computation is extremely efficient. Loops with carried dependencies can be unrolled into pipelines, and if the loops are not fully pipelined, the data can be interleaved to hide latency. The Java code is essentially a meta-program that describes the configuration of the dataflow engines [9].

The code is compiled and executed to create a `.max` file, which is a configuration file with the proper bitstream to configure the FPGA chip. The configuration file also contains various meta-information as well as CPU function calls for easier integration with the complete CPU application. The CPU code is handled the standard way, but it is linked with the `.max` file and a SLiC library (see Section 2.6.3) to connect CPU application and DFEs. The full, final executable includes all the binary code necessary to run the CPU application and perform computations on DFEs, which is done through a series of function calls. The DFEs are configured, any streams of data coming from the main memory is orchestrated, and computation is initiated. The compilation process can take a long time, typically hours or even days for larger programs, so development and testing is typically done through simulations. Simulations are run purely on the CPU, and are significantly slower, but can be compiled in seconds or minutes. This enables dataflow kernels, and entire dataflow programs, to be developed with a trial-and-error approach. The simulator gives insight into the computation and execution of kernels, but since it is slow, small amounts of data should be used for testing and verification before large amounts are run on the actual DFEs [10, 9].

There are two stages to compilation [9]:

1. The first stage is Java compilation since kernels and managers are written in Java. This stage is largely standard Java compilation, but the MaxCompiler has its own Java compiler because of the MaxJ extension.

2. The next stage, or rather stages, are done at runtime of the Java program, i.e. still part of the MaxCompiler complilation stage and not application runtime. The compiled Java code is executed to perform various operation:

a) **Graph construction**
This is the step that creates the dataflow graph for the computation. The graph is constructed in the memory according to the Kernel Compiler API.

b) **Kernel Compiler compilation**
The Kernel Compiler uses the generated graph to create a low-level representation that can then be used to generate a dataflow engine configuration or simulation model.

c) **Back-end compilation**
The back-end compilation is the final step that generates the DFE configurations. This step involves third-party tools to generate configurations for the specific chip that will emulate the DFE.

2.6.2 Implementing Kernels

Implementing a kernel is the process of translating the arithmetic for a given piece of code into a dataflow graph. Since Java is used, this involves mostly conventional classes, function calls, expressions etc. and common techniques like object-orientation can be utilized, which makes the development much easier compared to traditional digital design. However, this only applies to the meta-programming that helps to generates the hardware components, and not the computation that takes place upon execution, which has some restrictions, e.g. single-assignment. MaxCompiler includes operations from the digital design realm for connecting variables and concatenation. The way loops are handled is also different as they are always unrolled to create sequential code, rather than actually create conventional loops. The final kernel graphs for an application will contain all the units needed to implement the code on DFEs, using the following types of nodes [9]:

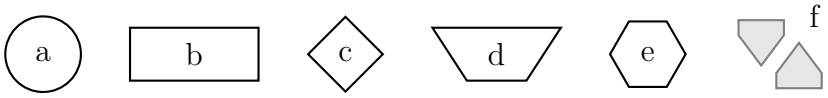


Figure 2.8: Types of nodes in dataflow graphs, adapted from *Multiscale Dataflow Programming, version 2016.1.1* [9]

Computation nodes (2.8a) are labelled with the arithmetic or logic operation which they perform. Typecasting is also represented with circular computation nodes.

Value nodes (2.8b) are values, or parameters, and can either be constant or set at runtime.

Stream offsets (2.8c) enable the design to look forward and backward in data streams to use past and future data elements.

Multiplexer nodes (2.8d) are used for making decisions, i.e. give conditional behaviour.

Counter nodes (2.8e) are used to direct control flow, e.g. length of rows/-columns in arrays and keeping track of the current position in a stream for boundary computations.

I/O nodes (2.8f) connect the data streams between kernels and the Manager.

2.6.3 SLiC Interface

As described in Section 2.6.1, a Maxeler computing system will run conventional executable files on a CPU and `.max` configuration files on DFEs. The `.max` files are loaded by the CPU, initialized and run on available DFEs, and this done through SLiC API functions. The functions execute *actions* on the DFEs, e.g. sending datastreams and parameters. By default the SLiC interface is accessed with C code, but there are also *SLiC skins* available so basic SLiC interface function calls can be made in other languages. There are three types of SLiC interfaces available, but in practice, all but the most basic application needs the advanced features of the SLiC API. The advanced features give greater control and easier debugging as the interface is explicitly configured and accessed instead of having to rely on automatically generated functions. The main programming challenge associated with the SLiC interface is calculating sizes of and handling datastreams. For standard data types with C-supported precision the process is a straight-forward mapping in CPU code and manager, but uncommon precision is often used for increased efficiency and better performance [9].

The three types of SLiC interfaces are [9]:

1. **Basic Static**
   Computation is done on a DFE through a single function call, which is automatically generated by the compiler. Static actions, number of streams, stream sizes etc. are defined for the particular `.max` file that is used.

2. **Advanced Static**
   This interface allows more control when loading DFEs. Multiple complex actions can be set, and the developer can optimize CPU and DFE collaboration.

3. **Advanced Dynamic**
   The dynamic interface supports all the available dataflow optimizations. The developer has fine-grained control of allocation of dataflow resources.

The life cycle of `.max` files in an application, whether advanced or basic features are used, is as follows [9]:

1. **Load**
   The `.max` file is loaded onto a DFE, after which the DFE will be exclusively owned by the calling CPU process. Loading `.max` files takes between 100 milliseconds and 1 second.

2. **Execute actions**
   The CPU uses SLiC interface functions to perform actions on the DFE. The loaded `.max` file should be utilized for long enough to make the long, relatively speaking, load time worth it.

3. **Unload**
   The CPU releases the DFE and returns it to the pool of DFEs, which is managed by the underlying MaxelerOS.

2.6.4 Dataflow Manager

The manager's work is to connect the kernels to the CPU application and orchestrate datastreams. It also handles compilation for either DFE execution or simulation. The manager is created by importing the `Manager` class, or

by extending the `CustomManager` class, and creating at the very least a `main` method to run the build process. In the `main` method a manager object is created using parameters in the form of a `EngineParameters` object, which determines whether the design is built for DFEs or simulation. This is controlled by *run rules* in the environment that is parsed by the MaxCompiler, but apart from the run rules and the resulting engine parameters, there are no differences between managers used for DFE execution and simulation. Advanced managers uses the `CustomManger` class, has a custom constructor, and additional methods, e.g. to create engine parameters [9].

2.6.5 Datastreams

The concept of streams of data lies at the core of dataflow computing, and properly accessing values in the streams is key for efficiency. The datastreams are viewed through *windows*, which are held in on-chip memory on DFEs to minimize data transfers. This is done through the stream offset nodes (see Section 2.6.2), and the windows are the range from the largest offset to the smallest offset. MaxCompiler has three types of stream offset: Static, variable and dynamic. Static offsets are fixed at compile-time (hard-coded), variable offsets are set at runtime before streams are processed, and dynamic offsets are set at runtime during stream processing. Static and variable offsets are set according to values or standard Java-variables and are fixed for the full duration of streams, while dynamic offsets use `DFEVars` and can change for each kernel tick. Dynamic offsets supports more advanced flows of data, but comes with some overhead compared to the fixed approaches, which can be highly optimized by the compiler [9].

2.6.6 Control Flow

Control in dataflow computing is implemented with counters, which can be thought of as the equivalent of loops in conventional computing. The counters keep track of where kernels are in the current datastreams as well as various levels of streaming and iterations. There are simple counters, advanced timers, and counter chains, which enable nesting behaviour. The simple counter is created with number of bits and optionally wrap point as arguments, and will increment the value by one for each kernel tick, i.e. per

element in the stream. The counter counts from zero to one less than the wrap point or max value before starting again. Advanced counters have more options, including start value, increment value, count mode and wrap mode. Both simple and advanced counters can be nested using counter chains, which are objects that control multiple counters. Counters are added to the counter chain object one by one, and the outer counters (first ones added) will increment for every full count of the inner counters [9].

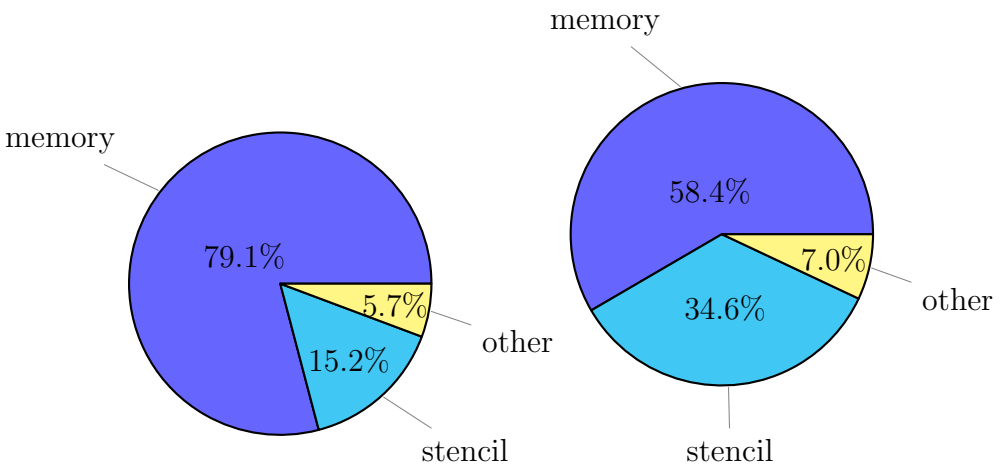
Chapter 3

Implementation

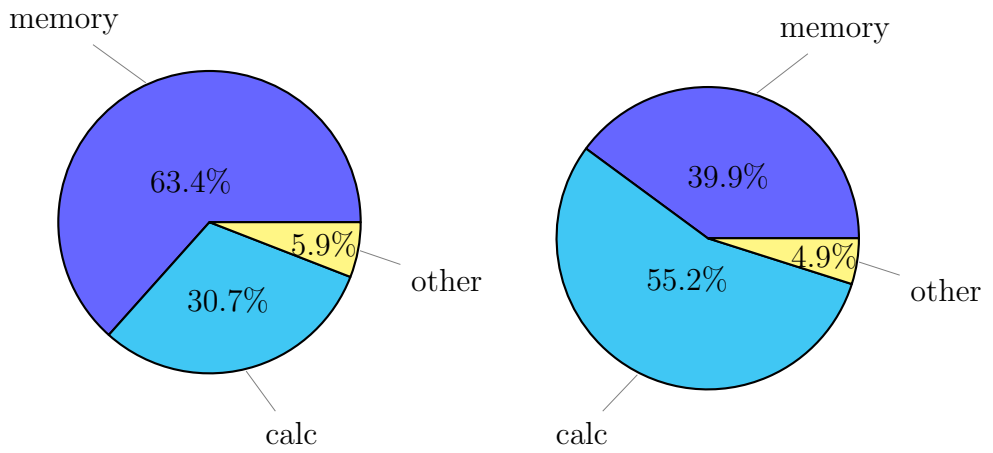
In this chapter we describe the process of accelerating miniAMR. We start by benchmarking and analysing the reference version to discover which parts of the application has the greatest potential for improvement. We go on to detail the necessary changes and the implementation of the dataflow version.

3.1 Performance Analysis

Figure 3.1 shows how different parts of miniAMR contributes to total execution time. It is clear that there are two main areas we need to focus on: Memory management and stencil calculation. That is, allocating and freeing the data used for the application, and the 7-point and 27-point stencil calculation, which lies at the core of the application. For small problem sized, memory management is the most critical part of the application, taking up 79.1 % of the total execution time when using the 7-point stencil and 63.4 % when using the 27-point stencil for a problem size of 10^3. For the 7-point stencil, memory management remains the most critical part for larger problem sizes, but when running the 27-point stencil, the stencil calculation eventually overtakes it, 55.2 % over 39.9 % for a problem size of 50^3.



(a) 7-point stencil, problem size 10^3 (b) 7-point stencil, problem size 50^3



(c) 27-point stencil, problem size 10^3 (d) 27-point stencil, problem size 50^3

Figure 3.1: Distribution of execution time for reference version of miniAMR

3.2 Memory Management

The data in miniAMR is double values stored four-dimensional arrays. The first dimension is the index for variables that will be calculated on and the rest of the dimensions are the indices for the actual three-dimensional data: x, y and z. The reference version of miniAMR uses a quadruple pointer and multiple nested loops to allocate and deallocate the data, which is inefficient, leads to fragmented memory and is ultimately what causes the memory part of the application to take up a lot of the total execution time. Additionally, this type of data structure is not suitable when interfacing the Maxeler system. The SLiC interface requires single pointers and data stored in contiguous memory.

To make the memory management more efficient we rewrite the application to operate with single pointers. This is done by changing the multiple nested allocation calls to a single large call for each variable, and similarly a single deallocation call. This ensures that the data is stored in contiguous memory and is also much faster, see Chapter 4. Additionally the contiguous memory means that the principle of spatial locality applies, which could potentially speed up the stencil calculation. To calculate the proper index from index variables in a similar fashion to standard multidimensional indexing, we create a function, which uses the problem size to calculate offsets in x, y and z directions.

```
1 size_t calc_index(int var, int x, int y, int z) {
2     return (var * x_size * y_size * z_size)
3          + (x * y_size * z_size)
4          + (y * z_size)
5          + x;
6 }
```

Listing 3.1: Function to calculate multidimension index

```
1 value = array[var][x][y][z];
2 value = array[calc_index(var, x, y, z)];
```

Listing 3.2: Multidimensional indexing using multiple pointers (line 1) and a single pointer (line 2)

3.3 Stencil Calculation

The stencils that lie at the core of miniAMR are basic averaging stencils: A 7-point star stencil and a 27-point cube stencil. The 7-point stencil (see Figure 3.2) uses elements from $\pm$ 1 in all directions, and the 27-point stencil is similar, but also requires all the edges and corners.

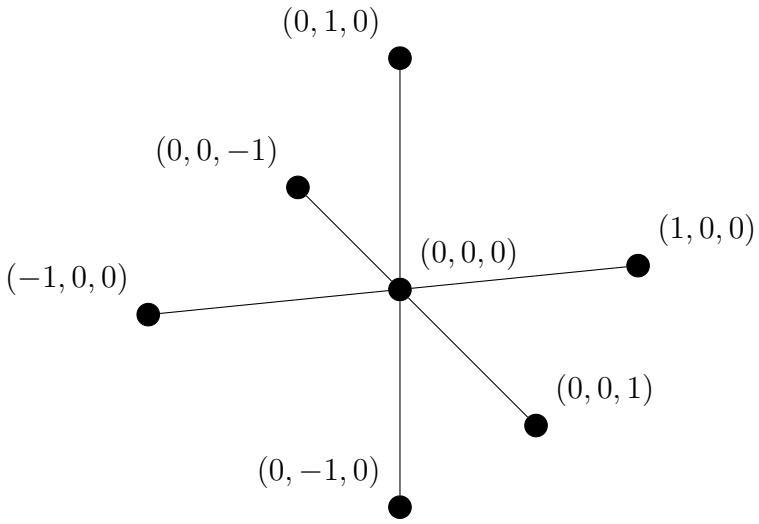


Figure 3.2: 7-point stencil used in miniAMR

Listing 3.3 shows 7-point stencil implemented in C. It uses three nested loops and has seven memory accesses, six sums and one division for each iteration. Moving this calculation onto DFEs involves three main challenges: Creating the datastream, getting all the seven elements from the datastream and performing the stencil calculation, and handling the edges. After rewriting the application to operate with single pointers, the datastreams are already laid out in contiguous memory and we can create new pointers by adding the offset for each array to the initial pointer. Running on multiple DFEs complicates matters somewhat (see section 3.4), but creating the datastreams only requires basic pointer arithmetic. Performing the stencil calculation and handling edges, however, require more work.

```
    for (i = 1; i <= x_size; i++) {
        for (j = 1; j <= y_size; j++) {
            for (k = 1; k <= z_size; k++) {
                work[i][j][k] = (array[i-1][j  ][k  ] +
                                 array[i  ][j-1][k  ] +
                                 array[i  ][j  ][k-1] +
                                 array[i  ][j  ][k  ] +
                                 array[i  ][j  ][k+1] +
                                 array[i  ][j+1][k  ] +
                                 array[i+1][j  ][k  ])/7.0;
            }
        }
    }
```

Listing 3.3: C code for 7-point stencil in three dimensions

3.3.1 Dataflow Kernel

The sum for the stencil can be split into multiple operations and parallelized since all the values required are separate and without data dependencies. When running CPU code, the compiler and CPU will handle this through optimization and multiple issue, out-of-order execution etc., but for dataflow kernels we have to implement this explicitly. Using a standard, naive expression like in Listing 3.3 leads to a serialized sum as shown in Figure 3.3.

The standard sum for the 7-point stencil takes six cycles to produce the result. By implementing partial sums running in parallel, we can reduce this to three cycles as shown in Figure 3.4. Partial sums works the same way for the 27-point stencil, except it's even more significant as 26 cycles are reduced to just five. Using partial sums like this has similar space requirements on the DFEs, and give increased parallelism and performance at the cost of code size and complexity for the developer as the sums need to be implemented specifically for the stencil used.

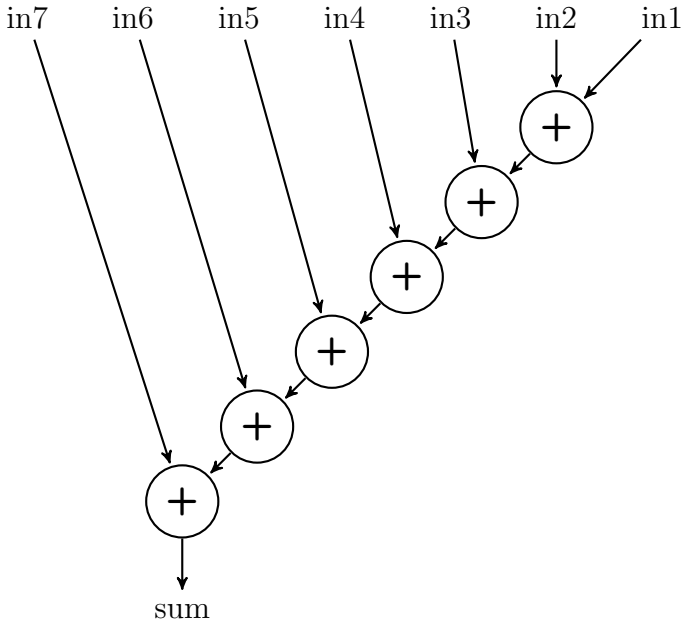


Figure 3.3: Dataflow graph for standard sum

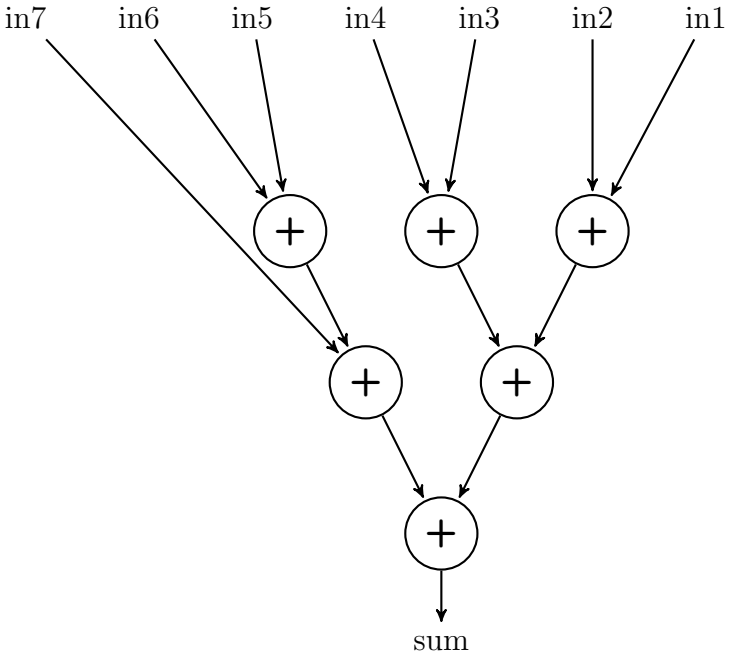


Figure 3.4: Dataflow graph for optimized sum

3.3.2 MaxJ Kernel Description

Listing 3.4 shows the kernel description in `MaxJ` for the 7-point stencil. We have created a function to generate the code, which makes it easy to implement multiple datastreams meaning we can work on multiple arrays in parallel. The function uses the input stream and offset expressions to get all the required data elements. The offset expressions are calculated and set in CPU code similarly to the index function in Listing 3.1 and passed through the manager to create variable stream offsets (see Section 2.6.5). This means that we can handle varying problem sizes without creating multiple kernels.

```
1  private DFEVar stencil(DFEVar input, OffsetExpr x_offset,
       OffsetExpr y_offset) {
2      DFEVar x1 = stream.offset(input, -x_offset);
3      DFEVar x3 = stream.offset(input, x_offset);
4      DFEVar y1 = stream.offset(input, -y_offset);
5      DFEVar y3 = stream.offset(input, y_offset);
6      DFEVar z1 = stream.offset(input, -1);
7      DFEVar z3 = stream.offset(input, 1);
8
9      DFEVar s1 = x1 + x3;
10     DFEVar s2 = y1 + y3;
11     DFEVar s3 = z1 + z3;
12
13     DFEVar s4 = s1 + s2;
14     DFEVar s5 = s3 + input;
15
16     DFEVar sum = s4 + s5;
17     DFEVar const_7 = constant.var(7.0);
18     DFEVar result = sum / const_7;
19     return result;
20 }
```

Listing 3.4: MaxJ code for 7-point stencil in three dimensions

Unlike the stencil written in C code, which only fetches the required elements from the array, our kernel operates on a continuous stream and we have to stream the whole array through the DFEs. This gives us the aforementioned challenge with handling the edges. For the edges of the arrays, the original input element should be passed to the output streams. We solve this by implementing three counters (see Section 2.6.6) for the x, y and z indices.

Using these counters we can detect when the current input element is an edge as shown in Listing 3.5 and use a multiplexer to choose whether the input or the result of the stencil calculation is passed to output.

```
1  DFEVar x_under = x.eq(0);
2  DFEVar x_over = x.eq(x_size-1);
3  DFEVar x_edge = x_under | x_over;
4  DFEVar y_under = y.eq(0);
5  DFEVar y_over = y.eq(y_size-1);
6  DFEVar y_edge = y_under | y_over;
7  DFEVar z_under = z.eq(0);
8  DFEVar z_over = z.eq(z_size-1);
9  DFEVar z_edge = z_under | z_over;
10 DFEVar edge = x_edge | y_edge | z_edge;
11
12 DFEVar output = edge ? input : result;
```

Listing 3.5: MaxJ code for boundary check

3.4 Running on Multiple DFEs

Our Maxeler system is a MAX3, which contains four Vectis DFEs. To achieve maximum performance we want to have all engines doing as much work in parallel as possible. The limit for number of input and output streams is eight per DFE so at most we can handle 32 datastreams at once [8]. The Maxeler system has functions for non-blocking execution and synchronization, so by checking the number of variables to calculate on, we can select a number of DFEs to use between one and four. We can then handle between eight and 32 streams on DFEs, potentially in multiple iterations for large numbers of variables, and then calculate the rest on the CPU. We have to synchronize the engines after each iteration to make sure we do not get overlapping datastreams, which would ruin the results. This leads to low utilization and idle engines for some iterations, but moves as much of the calculation as possible onto DFEs while still being able to handle all problem sizes.

We could potentially allocate additional memory space for all output streams from DFEs. This would enable us to run all iterations asynchronously, keep the utilization higher, and only synchronize the last iteration. However, this gives an significant increase in complexity in addition to raising the mem-

ory footprint, and might not lead to better performance. Managing multiple varying memory locations and keeping track of iterations and streams would require fundamental changes to miniAMR, and if the implementation requires copying or moving memory around, the overhead likely eats up any performance benefits gained from increased utilization.

Chapter 4

Results and Discussion

In this chapter we present the results, comment on their meaning, and discuss trade-offs, alternative solutions, shortcomings etc. We start with the memory section of the application and distribution of execution time before moving on to stencil calculation and total speedup. For the diagrams in this chapter the reference version of miniAMR is referred to as "ref", the rewritten version as "ref_fix" and the dataflow version as "dataflow".

4.1 Memory Management

Rewriting miniAMR to use single pointers reduces the execution time spent on allocation and deallocation to almost zero. Figure 4.1 shows the execution time in seconds that is spent on the memory section of the application for the reference version and the rewritten version of miniAMR. As single pointers are used, the number of allocation calls in the rewritten version is constant for all problem sizes and only depend on the top index, i.e. number of variables to calculate on. This translates to a near constant execution time. The reference version, on the other hand, shows exponential growth in both allocation calls and execution time.

After improving the memory section, the stencil calculation becomes the critical part of the application for both stencils and all problem sizes. As Figure 4.2 shows, the stencil calculation now stands for over 90 % of the

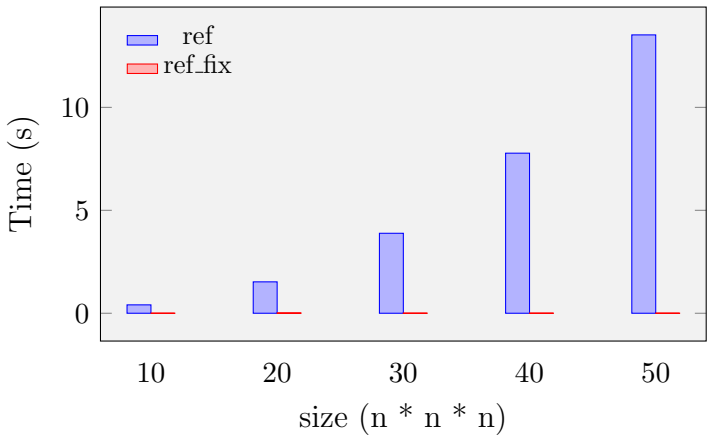
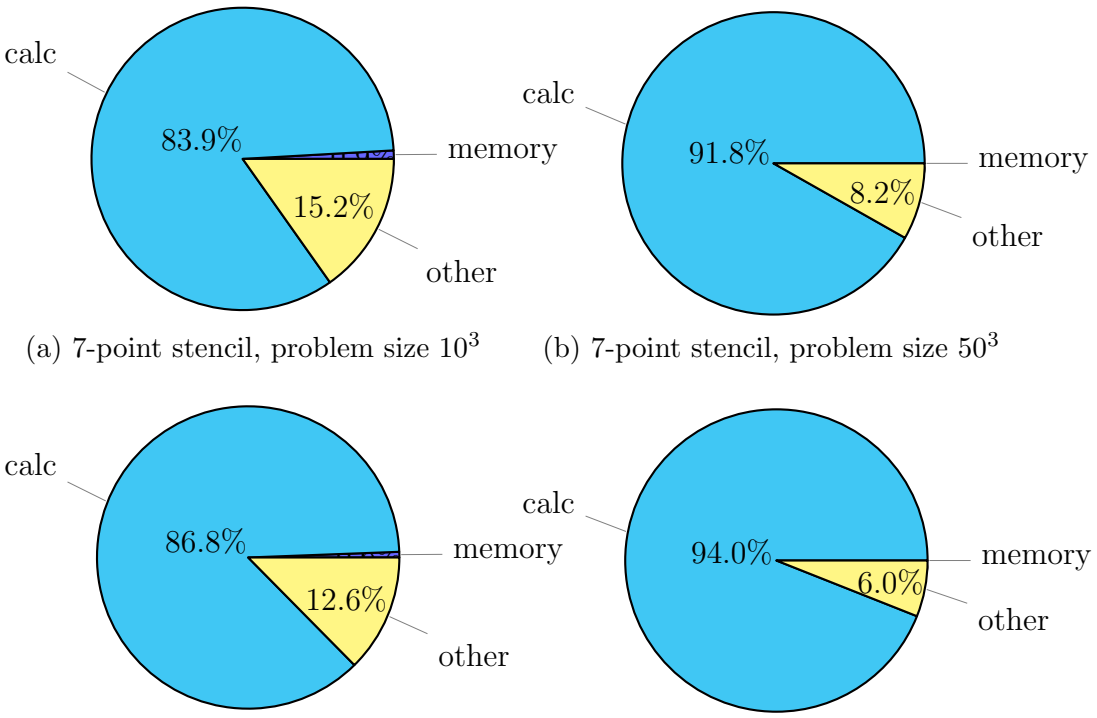


Figure 4.1: Execution time spent allocating and freeing data

execution time for larger poblem sizes, and is an excellent target for acceleration. When implementing the stencils on DFEs, we add some overhead, but we are able to keep the percentage over 80 (see Figure 4.3). The overhead stems from loading maxfiles and preparing the actions which run on DFEs. This overhead is constant for both stencils regardless of problem size as it does not depend on the number of data elements, but rather the kernels and number of DFEs used.

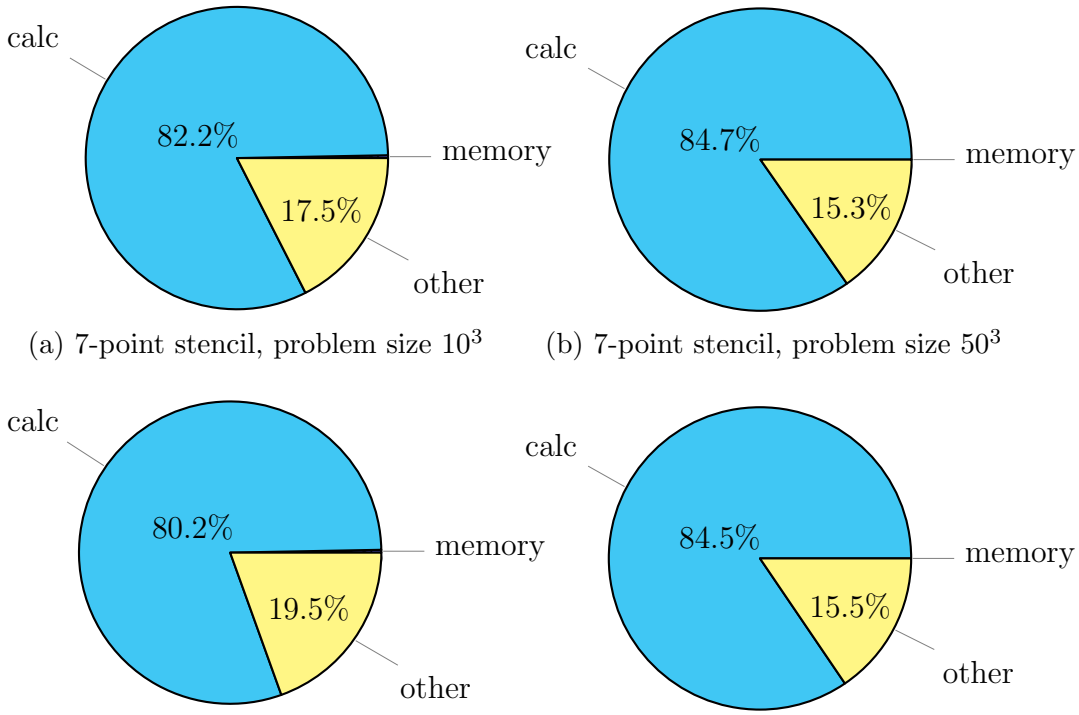
Unlike the reference and rewritten versions of miniAMR, the distributions of execution time is roughly the same for the 7-point and 27-point stencils for the dataflow version. The 27-point stencil require some additional cycles for partial sums, five vs. three (see Section 3.3.1), but the additional elements can all be laid out in space on the DFEs. Listing 4.1 and Listing 4.2 Show the total resource usage on the DFEs for the dataflow kernels. We want to utilize as much of the chips as possible for maximum performance, and the 27-point kernel covers over 80 % of available logic resources. This results in a significant speedup, which we will show in Section 4.2.



(a) 7-point stencil, problem size 10^3 (b) 7-point stencil, problem size 50^3

(c) 27-point stencil, problem size 10^3 (d) 27-point stencil, problem size 50^3

Figure 4.2: Distribution of execution time for rewritten version of miniAMR



(a) 7-point stencil, problem size 10^3 (b) 7-point stencil, problem size 50^3

(c) 27-point stencil, problem size 10^3 (d) 27-point stencil, problem size 50^3

Figure 4.3: Distribution of execution time for dataflow version of miniAMR

Listing 4.1: Resource usage for 7-point stencil dataflow kernel

```
FINAL RESOURCE USAGE
Logic utilization:          89497 / 297600 (30.07%)
    LUTs:                   78267 / 297600 (26.30%)
    Primary FFs:            80570 / 297600 (27.07%)
    Secondary FFs:          13478 / 297600 (4.53%)
Multipliers (25x18):            0 / 2016   (0.00%)
    DSP blocks:                 0 / 2016   (0.00%)
Block memory (BRAM18):        395 / 2128   (18.56%)
```

Listing 4.2: Resource usage for 27-point stencil dataflow kernel

```
FINAL RESOURCE USAGE
Logic utilization:         242175 / 297600 (81.38%)
    LUTs:                  211391 / 297600 (71.03%)
    Primary FFs:           217499 / 297600 (73.08%)
    Secondary FFs:          48170 / 297600 (16.19%)
Multipliers (25x18):            0 / 2016   (0.00%)
    DSP blocks:                 0 / 2016   (0.00%)
Block memory (BRAM18):        410 / 2128   (19.27%)
```

4.2 Speedup

Figure 4.4 and Figure 4.5 show the total speedup for the 7-point and 27-point stencils, respectively, running problem sizes between 10^3 and 50^3. The speedups are calculated by comparing against the reference version of miniAMR. For both stencils, the rewritten reference version is the fastest for smaller problem sizes while the dataflow version comes out on top when the problem size grows. We can also observe that the speedup of rewritten version shrinks for larger problem sizes. This is because a function is used to calculate indices for every single access to the arrays that store the data, which is necessary since the problem size is set at runtime, but adds significant overhead as the problem size grows. This overhead is avoided when the stencil calculation is moved onto DFEs.

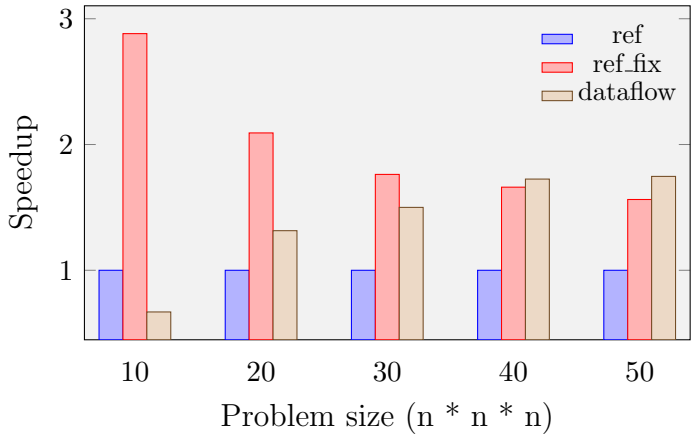


Figure 4.4: Speedup for the different versions of miniAMR running the 7-point stencil

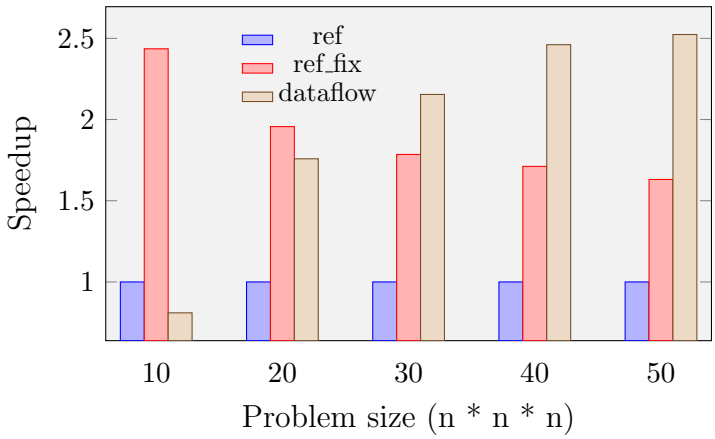


Figure 4.5: Speedup for the different versions of miniAMR running the 27-point stencil

In theory we could have thousands of dataflow cores, i.e. functional running concurrently, but we are limited by the number of streams that the DFEs support and potentially I/O bandwidth as the number of streams increases. However, we see significant speedup of over 1.5 for larger problem sizes as the raw throughput is much higher than what the CPU can provide. This is most visible for the 27-point stencil is it properly utilizes the concept of computation in space and covers over 80 % the available logic resources to give us a speedup of over 2.5.

Chapter 5

Conclusion and Future Work

In this chapter we comment we conclude the project and propose topics for future work. We describe our results, list potential improvements for the dataflow version of miniAMR and comment on the reusability of dataflow kernels and implementation code.

5.1 Conclusion

Maxeler's Multiscale Dataflow Computing system eases the typically difficult and time-consuming FPGA development process. This is important as we observe diminishing returns from traditional parallel computation and increased interest in FPGA-based acceleration. Using the Maxeler system, we have altered miniAMR, a proxy application for adaptive mesh refinement developed by Mantevo project, to run on DFEs. By properly managing memory and moving the core 3D stencil computation onto DFEs we achieve a maximum speedup of 2.52 while maintaining all functionality of the application. For small problem sizes the dataflow version cannot match CPU execution because of reduced utilization and being limited to only eight datastreams per engine, but for larger problem sizes the increased throughput gives a significant increase in performance.

5.2 Future Work

The kernels we have developed in this project primarily use datastreams from CPU to DFE. The DFEs have a complete memory system, which could be used, but we chose not to do this as the streams from DFE memory have less flexibility, and initial efforts showed little promise because miniAMR operates with a large number of small arrays rather than a large array. The maximum number of streams DFE memory is 16, compared to eight from the CPU, which means a greater potential for parallelism. For future work, we suggest making multiple specialised memory-driven kernels, e.g. for a certain number of inputs, and testing whether the overhead from loading maxfiles or reduced utilization from having certain engines idle is worth it.

Since the Maxeler system uses a high-level approach to development, the kernels from this project are reusable and adaptable. Initializing the DFEs is relatively simple through the SLiC interface and we have created functions and loops for our kernels, which means that implementation in other applications should take minimal effort as the initial development effort has already been done. We suggest implementing the kernels in applications that are similar to miniAMR or use 3D stencils at the core of computation to test the efforts required and speedup that can be achieved without having to significantly alter the kernels.

References

[1] Jacob A Bower et al. "A java-based system for fpga programming". In: *FPGA World Conference*. 2008.

[2] Pong P Chu. *FPGA prototyping by VHDL examples: Xilinx Spartan-3 version*. John Wiley & Sons, 2008.

[3] Erik D'Hollander and Kristof Beyls. "High performance computing with FPGAs". eng. In: *High Speed and Large Scale Scientific Computing*. Ed. by Wolfgang Gentzsch, Lucio Grandinetti, and Gerhard Joubert. Vol. 18. Advances in Parallel Computing. IOS Press, 2009, pp. 55–73. ISBN: 9781607500735. DOI: 10.3233/978-1-60750-073-5-55.

[4] M. C. Herbordt et al. "Achieving High Performance with FPGA-Based Computing". In: *Computer* 40.3 (Mar. 2007), pp. 50–57. ISSN: 0018-9162. DOI: 10.1109/MC.2007.79.

[5] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. "Advances in Dataflow Programming Languages". In: *ACM Comput. Surv.* 36.1 (Mar. 2004), pp. 1–34. ISSN: 0360-0300. DOI: 10.1145/1013208.1013209.

[6] Olav Lindtjorn et al. "Surviving the end of scaling of traditional microprocessors in HPC". In: *Proceedings of HOT CHIPS*. Vol. 22. 2010.

[7] Mantevo.org. *Mantevo Project*. 2018. URL: https://mantevo.org/ (visited on 08/19/2018).

[8] *MaxCompiler Manager Compiler Tutorial, version 2016.1.1*. Maxeler Technologies.

[9] *Multiscale Dataflow Programming, version 2016.1.1*. Maxeler Technologies.

[10] Oliver Pell et al. "Maximum Performance Computing with Dataflow Engines". In: *High-Performance Computing Using FPGAs*. Ed. by Wim Vanderbauwhede and Khaled Benkrid. New York, NY: Springer New

York, 2013, pp. 747–774. ISBN: 978-1-4614-1791-0. DOI: 10.1007/978-1-4614-1791-0_25.

[11] Khaled Benkrid Wim Vanderbauwhede, ed. *High-Performance Computing Using FPGAs*. Springer, New York, NY, 2013. ISBN: 978-1-4614-1791-0.