Linköping Studies in Science and Technology
Dissertations No. 1641

Efficient Compilation for Application Specific Instruction set DSP Processors with Multi-bank Memories

Joar Sohl

Department of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden

Linköping 2015

ISBN 978-91-7519-151-5

ISSN 0345-7524

Efficient Compilation for Application Specific
Instruction set DSP Processors with Multi-bank Memories
Joar Sohl
ISBN 978-91-7519-151-5

Copyright © Joar Sohl, 2015

Linköping Studies in Science and Technology
Dissertation No. 1641
ISSN 0345-7524

Department of Electrical Engineering
Linköping University
SE-581 83 Linköping
Sweden
Phone: +46 13 28 10 00

Author e-mail: `joar@sohl.se`

Cover image

A colored graph.

Printed by UniTryck, Linköping University
Linköping, Sweden, 2015

Abstract

Modern signal processing systems require more and more processing capacity as times goes on. Previously, large increases in speed and power efficiency have come from process technology improvements. However, lately the gain from process improvements have been greatly reduced. Currently, the way forward for high-performance systems is to use specialized hardware and/or parallel designs.

Application Specific Integrated Circuits (ASICs) have long been used to accelerate the processing of tasks that are too computationally heavy for more general processors. The problem with ASICs is that they are costly to develop and verify, and the product life time can be limited with newer standards. Since they are very specific the applicable domain is very narrow.

More general processors are more flexible and can easily adapt to perform the functions of ASIC based designs. However, the generality comes with a performance cost that renders general designs unusable for some tasks. The question then becomes, how general can a processor be while still being power efficient and fast enough for some particular domain?

Application Specific Instruction set Processors (ASIPs) are processors that target a specific application domain, and can offer enough performance with power efficiency and silicon cost that is comparable to ASICs. The flexibility allows for the same hardware design to be used over several system designs, and also for multiple functions in the same system, if some functions are not used simultaneously.

One problem with ASIPs is that they are more difficult to program than a general purpose processor, given that we want efficient software. Utilizing all of the features that give an ASIP its performance advantage can be difficult at times, and new tools and methods for programming them are needed.

This thesis will present ePUMA (embedded Parallel DSP platform with Unique Memory Access), an ASIP architecture that targets algorithms with predictable data access. These kinds of algorithms are very common in e.g. baseband processing or multimedia applications. The primary focus will be on the specific features of ePUMA that are utilized to achieve high performance, and how it is possible to automatically utilize them using tools. The most significant features include data permutation for conflict-free data access, and utilization of address generation features for overhead free code execution. This sometimes requires specific information; for example the exact sequences of addresses in memory that are accessed, or that some operations may be performed in parallel. This is not always available when writing code using the traditional way with traditional languages, e.g. C, as extracting this information is still a very active research topic. In the near future at least, the way that software is written needs to change to exploit all hardware features, but in many cases in a positive way. Often the problem with current methods is that code is overly specific, and that a more general abstractions are actually easier to generate code from.

Preface

This thesis includes material from the following papers.

I am the primary author of the following papers:

- **Joar Sohl**, Andréas Karlsson, Dake Liu, "Efficient Vector Processing with an Application Specific Instruction set Processor", Manuscript.
- **Joar Sohl**, Andréas Karlsson, Dake Liu, "Efficient Compilation for DSP Architectures with 2D-Addressing", Manuscript.
- **Joar Sohl**, Andréas Karlsson, Dake Liu, "Conflict-free data access for multi-bank memory architectures using padding", High Performance Computing (HiPC), 2013, 425-432, 2013.
- **Joar Sohl**, Jian Wang, Andréas Karlsson, and Dake Liu. "Automatic permutation for arbitrary static access patterns". In Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on, pages 215-222, july 2012.
- **Joar Sohl**, Jian Wang, and Dake Liu. "Large matrix multiplication on a novel heterogeneous parallel DSP architecture". In Proceedings of the 8th International Symposium on Advanced Parallel Processing Technologies, APPT '09, pages 408-419, Berlin, Heidelberg, 2009. Springer-Verlag.

In addition, I am the co-author of the following papers:

- Dake Liu, **Joar Sohl**, and Jian Wang. "Parallel Programming and Its Architectures Based on Data Access Separated Algorithm Kernels".

International Journal of Embedded and Real-Time Communication Systems, 1(1): 65-85, 2010.

- Andréas Karlsson, **Joar Sohl**, Jian Wang, Dake Liu, "ePUMA: A unique memory access based parallel DSP processor for SDR and CR", Global Conference on Signal and Information Processing (GlobalSIP), 2013 IEEE, 1234-1237, 2013.
- Jian Wang, Andréas Karlsson, **Joar Sohl**, and Dake Liu. "Convolutional decoding on deep-pipelined SIMD processor with flexible parallel memory". In Digital System Design (DSD), 2012, pages 529-532. IEEE, 2012.
- Dake Liu, Andréas Karlsson, **Joar Sohl**, Jian Wang, Magnus Pettersson, and Wenbiao Zhou. "ePUMA embedded parallel DSP processor with unique memory access". In Information, Communications and Signal Processing (ICICS) 2011 8th International Conference on, pages 1-5, dec. 2011.
- Jian Wang, Andréas Karlsson, **Joar Sohl**, Magnus Pettersson, and D. Liu. "A multi-level arbitration and topology free streaming network for chip multiprocessor". In ASIC (ASICON), 2011 IEEE 9th International Conference on, pages 153-158, oct. 2011.
- Jian Wang, **Joar Sohl**, Andréas Karlsson, and Dake Liu. "An efficient streaming star network for multi-core parallel DSP processor". In Proceedings of the 2011 Second International Conference on Networking and Computing, ICNC '11, pages 332-336, Washington, DC, USA, 2011. IEEE Computer Society.
- Erik Hansson, **Joar Sohl**, Christoph Kessler, and Dake Liu. "Case Study of Efficient Parallel Memory Access Programming for the Embedded Heterogeneous Multicore DSP Architecture ePUMA", Proc. Int. Workshop on Multi-Core Computing Systems (MuCoCoS-2011), June 2011, Seoul, Korea. IEEE CS Press.

- Jian Wang, **Joar Sohl**, and Dake Liu. "Architectural support for reducing parallel processing overhead in an embedded multiprocessor". In Proceedings of the 2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, EUC '10, pages 47-52, Washington, DC, USA, 2010. IEEE Computer Society.
- Jian Wang, **Joar Sohl**, Olof Kraigher, and Dake Liu. "Software programmable data allocation in multi-bank memory of SIMD processors". In Proceedings of the 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools, pages 28-33. IEEE Computer Society, 2010.
- Jian Wang, **Joar Sohl**, Olof Kraigher, and Dake Liu. "ePUMA: A novel embedded parallel DSP platform for predictable computing". In Education Technology and Computer (ICETC), 2010 2nd International Conference on, volume 5, pages V5-32 -V5-35, june 2010.

Contributions

The major contributions can be summarized as follows:

Conflict-free memory access Evaluation of earlier methods for conflict-free access and formalization and implementation of new suitable methods for a more general DSP platform with multi-bank memories and permutation networks.

Multi-bank memories with permutation networks Evaluation of using multi-bank memories with permutation w.r.t. performance.

Efficient compilation of addressing Formalization and evaluation of a method of compiling vector based code that can significantly reduce the overhead of control code.

Programming Model Formalization of a programming model suitable for scratch-pad multi-bank memories, based on the characteristics of these systems and possibilities of automation during software development.

I have also contributed with the following software tools:

Memory resolver A tool for allocating logical memory addresses to physical addresses in such a way that memory bank conflicts are minimized, using different methods depending on data access requirements.

Assembler Developed and maintained the (high level) assembler for the project. Due to the fact that it does not simply translate instructions it is implemented more or less as a standard compiler.

Linker/loader A tool for linking the executable code for ePUMA system and the Matrix Processing Elements (MPEs), and for loading the code for execution in e.g. the simulators.

Scheduler A tool that schedules tasks on the processor cores w.r.t. computation and data transfer time.

FT A domain specific language for writing computation kernels for ePUMA. It is based around vector operations and solves the more problematic parts of writing MPE code. E.g. allocating data buffers to memory, using address registers in an optimal way etc.

Data Access Template Configurator A tool for inserting computation code into already written templates for data access patterns.

Configurator A tool that instantiates the correct templates for given function calls.

Template assembler A tool that allows for generating assembly code based on hardware and function call parameters.

Acknowledgments

There are many people that I would like to thank for helping me during my time at Linköping University.

- Professor Dake Liu for giving me the opportunity to be a part of the ePUMA team, his support and supervision during my time at the division, and enthusiasm and commitment to the project.
- Andréas Karlsson and Dr. Jian Wang, my fellow ePUMA project members, for all the work they've put into the project.
- All the other Ph.D. students for all their comments and ideas regarding the ePUMA project.
- Ylva Jernling, Anita Kilander, and Gunnel Hässler for all their help in making even the most complicated administrative tasks simple.
- Everyone at the Computer Engineering division, past and present, for creating a great atmosphere and a great place to work.

This research was supported by the Swedish Foundation for Strategic Research (SSF).

Joar Sohl
Linköping, January 2015

Contents

Abbreviations	xix
I Background	1
1 Introduction	3
1.1 Motivation	4
1.2 Goals and Scope	5
1.3 Challenges	7
1.3.1 Multi-cores with Distributed Scratch-Pad Memory	7
1.3.2 Zero Overhead Execution	7
1.3.3 Parallel Memory Access	8
1.4 Method	10
1.4.1 Evaluation of Software Tools	11
1.4.2 Summary	12
1.5 Thesis Organization	12
2 ePUMA	15
2.1 Application Characteristics	15
2.1.1 Multimedia	15
2.1.2 SDR	16
2.2 ePUMA System Overview	17
2.3 Network on Chip	20
2.3.1 Distributed vs. Shared Memory	21
2.4 Matrix Processing Elements	21

2.4.1	Application Specific Instructions	22
2.4.2	MPEs - Data Level Parallelism using SIMD	23
2.4.3	Parallel Memories & Conflict-free Data Access	24
2.5	Architecture comparison	25
2.5.1	Imagine	26
2.5.2	VIRAM	26
2.5.3	CGRA	27
2.5.4	SODA/Ardbeg/AnySP	27
2.5.5	EVP	28
2.5.6	BBP	28
2.6	Summary	29
3	Optimization	31
3.1	Linear Programming	31
3.2	(Mixed) Integer Linear Programming	32
3.2.1	Exact Solution Strategies	32
3.2.2	Benefits	34
3.2.3	Drawbacks	35
3.3	Constraint Programming	36
3.3.1	Definition	36
3.3.2	Constraint Propagation	36
3.3.3	Example	37
3.3.4	Labelling	39
3.3.5	Optimization	39
3.3.6	Useful Constraints	39
3.3.7	Benefits	40
3.3.8	Drawbacks	40
3.4	Other methods	40
3.5	Summary	41
4	Dataflow analysis	43
4.1	Purpose	43
4.2	Example program	43
4.3	Summary	47

II Compilation Techniques 49

5 Scheduling 51

5.1 Problem/Model Definition	52
5.1.1 Events	52
5.1.2 Resources	52
5.1.3 Activities	53
5.1.4 Data blocks	54
5.1.5 Transfers	55
5.1.6 Calls	56
5.1.7 Objective Function	57
5.1.8 Partitioning and Symmetry Breaking	57
5.2 Conclusions	58

6 Conflict-free Memory Access 59

6.1 Motivation	59
6.2 Related Work	61
6.2.1 Why Multi-bank Memories?	62
6.3 Permutation	62
6.3.1 ILP model	66
6.3.2 Pre-processing	67
6.3.3 Post-processing	68
6.3.4 Applicability and Other Methods	69
6.4 Padding	70
6.4.1 Problem Formulation	71
6.4.2 Common Patterns	72
6.4.3 Generators for $\mathbb{Z}_n$	73
6.4.4 Padding for Straight-Line Access	73
6.4.5 Padding for Multiple Straight-Line Accesses	75
6.4.6 Padding Set Evaluation	75
6.4.7 Generating Valid Padding Sets	78
6.5 Rotation	79
6.6 Using Non-power of 2 Number of Memory Banks	81
6.7 Using Dynamic Bank Assignment Bits	82

6.8	Permutation of Dimensions	82
6.9	Summary	83
7	Code Generation for MPEs	85
7.1	FT	86
7.1.1	FT Features	87
7.2	Memory Management	89
7.2.1	Port Mapping	89
7.2.2	Variable Address Allocation	90
7.3	Vector Instruction Utilization	91
7.3.1	Problem Definition	92
7.3.2	Dataflow Analysis	96
7.3.3	General Idea	96
7.4	Transfer & Merge Functions	99
7.4.1	Address register requirements	99
7.4.2	Propagation of definitions	100
7.4.3	Other optimizations	101
7.5	Summary	103
III	Software Development for ePUMA	105
8	Programming Model	107
8.1	General programming flow	107
8.2	Kernel level development	108
8.2.1	Blocks	110
8.2.2	Pipelining	111
8.2.3	Block composition	112
8.2.4	Kernel templates	113
8.2.5	Data access templates	114
8.3	Discussion & Summary	118
9	Toolchain Support	119
9.1	Scheduler	119
9.2	Memory Resolver	120

9.3	FT Compiler	121
9.3.1	Data placement	123
9.3.2	Other Stages	123
9.4	mpeasm	124
9.4.1	Assembler Inputs	124
9.4.2	Assembler Outputs	125
9.4.3	Semantic analysis	126
9.4.4	Optimization	126
9.5	linker/loader	127
9.6	Data Access Template Configurator	128
9.7	Configurator	129
9.8	Toolchain flow	130

IV Results and Conclusions 133

10 Evaluation 135

10.1	Permutation	136
10.1.1	Preprocessing Performance	136
10.1.2	M-JPEG	139
10.1.3	FFT	142
10.2	Padding	145
10.2.1	Sorting	145
10.2.2	FFTs	149
10.2.3	FFTs including Non-power 2 FFTs	153
10.3	Vector Code Compilation	156
10.3.1	FIR filter	156
10.3.2	Matrix multiplication	159
10.3.3	Multiple matrix multiplications	162
10.3.4	2D convolution	165
10.4	LTE Downlink	168
10.5	Summary	169

11	Conclusions and Future Work	171
11.1	Programming Model	171
11.2	Conflict-free Data Access	171
11.3	MPE compilation	172
11.4	Summary	172
11.5	Future Work	172
12	Retrospectives	175
12.1	Scope	175
12.2	Number of hardware iterations	176
12.3	Order of work	177
12.4	Final remarks	178
	Bibliography	179

Abbreviations

ACC	Accelerator
AGU	Address Generation Unit
ALU	Arithmetic and Logic Unit
APE	Application Processing Element
API	Application Programming Interface
ASIP	Application Specific Instruction Set Processor
BBP	Baseband Processor
CC	Compute Cluster
CMP	Chip Multiprocessor
CP	Constraint Programming
CLP	Constraint Logic Programming
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DAG	Directed Acyclic Graph
DCT	Discrete Cosine Transform
DLP	Data Level Parallelism
DMA	Direct Memory Access
DSL	Domain Specific Language
DSP	Digital Signal Processing or Digital Signal Processor
FFT	Fast Fourier Transform
FIFO	First In First Out
FIR	Finite Impulse Response
GOPS	Giga Operations Per Second
GPS	Global Positioning System

GPU	Graphics Processing Unit
IDCT	Inverse Discrete Cosine Transform
IFFT	Inverse Fast Fourier Transform
ILP	Instruction Level Parallelism or Integer Linear Programming
IR	Intermediate Representation
ISA	Instruction Set Architecture
LSB	Least Significant Bits
LTE	Long Term Evolution
LUT	Lookup Table
LVM	Local Vector Memory
MAC	Multiply and Accumulate
MIMO	Multiple-Input and Multiple-Output
MIPS	Million Instructions Per Second
MPE	Matrix Processing Element
MM	Main Memory
NI	Network Interface
NOC	Network on Chip
PM	Program Memory
PT	Permutation Table
PVRF	Permutation Vector Register File
RAM	Random-Access Memory
RISC	Reduced Instruction Set Computing
SDR	Software Defined Radio
SDRAM	Synchronous Dynamic Random Access Memory
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Stream Multiple Thread
SPM	Scratch Pad Memory
TCM	Tightly Coupled Memory
TLP	Task Level Parallelism
VAGU	Vector Address Generation Unit
VLIW	Very Long Instruction Word
VMAC	Vector Multiply and Accumulate

Part I

Background

Chapter 1

Introduction

Digital Signal Processing (DSP) has become prevalent in many of the products we use on a daily basis, with perhaps the mobile phone as one of the most prominent examples. With it we can make telephone calls, surf the web, capture images/videos, use GPS navigation, play games etc. Even radar is starting to be used in consumer products such as cars, along with other image analysis to assist when driving.

A signal processing system in the most general form could be represented as in Figure 1.1. An input signal is processed in some way, and a useful output signal is extracted.

Figure 1.1: Signal processing system.

Many signal processing applications require a lot of computation. One such example is modern smart phones which require over 100 GOPS/W [1, 2]. For embedded DSP systems, reaching the performance requirements is not trivial. It is not just the problem of achieving sufficient performance at low enough power consumption, but the hardware implementation cost must be kept low as well. This problem is only exacer-

bated by the need to support a wider range of products and more complex standards. Using the traditional approach of just adding hardware and assuming that newer process technology will reduce the hardware cost and power requirements is no longer feasible.

Another problem is the longevity of different hardware modules. Developing and verifying hardware is expensive, so developing custom designs for one specific use case is not desirable. We would like to be able to reuse the same hardware design in several products and, if possible, use the same hardware for different functions in a system by using hardware multiplexing. I.e. if we have tasks that do not need to run in parallel and a flexible enough hardware block we could reuse the same hardware block for multiple purposes in a system design, saving silicon cost.

The problem we face is that for more general designs, the area cost and power consumption is higher than for specialized designs. On the other hand the overall design cost per product is lower.

1.1 Motivation

One way to deal with the increasing demands on embedded systems is to use an Application Specific Instruction set Processor (ASIP), a processor that have been specialized for a particular application domain, e.g. baseband processing. While not being as useful for tasks outside the application domain, the programmability compared to an ASIC gives it some very good properties.

Product lifetime A programmable solution could be used for several generations of different standards. E.g. one could possibly use the same chip for H.265 as for H.264. However, as performance requirements usually increase with newer standards this is obviously not always possible.

Flexibility A programmable solution also allows us to use the same processor in e.g. mobile phones or TVs. Flexibility here is when compared to an ASIC, and not other processors.

Chip volume An ASIP may be used in a wide range of products, and also over a longer time period than ASIC solutions. Consequently, the cost per chip can be reduced.

Development time It is easier to develop and debug software than hardware.

Upgrades and bug-fixes A programmable solution enables the devices to be upgraded, e.g. to support new video codecs. It also allows for correcting bugs, which reduces the verification requirements.

Hardware utilization/Cost As an ASIP can be used for multiple purposes it is possible to reduce the amount of inactive silicon. E.g. a chip could include one ASIP instead of one custom decoder for each video standard that should be supported. This gives a high hardware utilization, and consequently lower cost as a lower amount of silicon is required.

1.2 Goals and Scope

The goal of the ePUMA project [3] is to develop a flexible architecture for future DSP platforms, e.g. media and baseband. This includes the actual hardware platform and the software tools required to program it. In addition, it requires the implementation of a sufficient set of benchmarks to prove ePUMA's efficacy.

This thesis is mostly concerned with the software tools necessary to support the software development process for application programmers and kernel programmers. The initial idea when the ePUMA project was started was to have a flow similar to the process illustrated in Figure 1.2.

Given some existing code, the first step is to analyze what sections of code that could possibly be replaced by more efficient implementation. Secondly, when these sections have been identified, the next step is to choose an appropriate kernel. A kernel is a segment of code that computes some computationally expensive algorithm, e.g. FFT. This is

Figure 1.2: Programming flow.

a quite difficult problem to automate, however some progress has been made by another division involved in the same project [4].

The scope of this thesis is from the kernel configuration block to linking stage. That is, given a kernel, how can we:

- configure it so that we remove unnecessary overheads,
- schedule the computation as efficiently as possible,
- allocate data in such a way as to remove access overhead, and
- utilize the hardware features of the Matrix Processing Elements (MPEs), that are the cores used for the heavy processing, as efficiently as possible.

1.3 Challenges

There are quite a few challenges that complicate the compilation stages for the ePUMA platform, which will be described in more detail in Chapter 2.

1.3.1 Multi-cores with Distributed Scratch-Pad Memory

ePUMA has several compute clusters with their own scratch-pad memories rather than caches. The reasoning for this is that many DSP algorithms do not have good temporal locality, as they are often written and read only once. The accesses are in many cases not linear either. This makes it difficult to find a good fetching strategy to use for caches. It is quite common with fixed strides, and this can lead to unnecessary congestion and memory utilization by transferring and storing unused data in the cache lines. On the other hand the access is often predictable, and we can use software to control the data transfers. This is of course a burden on the programmers. Not only do we need to schedule the computation, but we must also manage the data transfers.

1.3.2 Zero Overhead Execution

The goal when designing an ASIP is that it should be efficient. Ideally we would like a processor core to issue one instruction every clock cycle that

contributes to the actual computed result.¹ However, overheads such as data access and controlling the flow of compute instructions may steal cycles and lower compute performance. Fortunately, the computation, data access, and control flow are often independent for DSP algorithms. The ePUMA platform provides very efficient addressing and several control flow mechanisms to overlap computing, data access, and control in time, but utilizing these manually is very cumbersome and error prone.

1.3.3 Parallel Memory Access

The purpose of an ASIP is to execute code for specific domains fast. For an ASIC design we might have something like Figure 1.3, where we have some memories and hardware to compute e.g. an n -point FFT.² For an ASIP, the primitive between memory for memory operations is usually smaller (at least on a hardware level), e.g. we only have smaller butterfly instructions available, as in Figure 1.4, and we are likely to process things layer by layer. The amount of data that we can read and write to these memories will determine a limit to how fast we may compute an algorithm. Ensuring that we can reach this limit is one of the major goals in the ePUMA project.

The major difficulty for efficient execution on the MPEs is parallel data access. Regardless of method of that we use, e.g. SIMD (as in ePUMA or Cell [5]) or VLIW [6], if we wish to compute things in parallel we require reading many values from memory in parallel, otherwise the resulting sequential parts will limit our performance [7]. It is often the case that these addresses are not consecutive, or aligned to the native vector width. A trivial way to solve it is to use memories with as many ports as necessary, however this has a prohibitive area cost. In ePUMA we use parallel vector memories that consists of several single-port memories that together make up the memory, coupled with a permutation network that enables access to parallel memory with a vector of addresses, given

¹For a single-issue machine.

²Most likely, the FFT block includes different memories or registers and performs the computation layer by layer, but from a user interface it looks something like this.

Figure 1.3: ASIC block.

that the addresses belong to different memory banks. This permutation network is also beneficial in that we will also reduce the need to issue shuffle instructions that often causes significant overheads, due to the fact that the memory access with permutation is performed before the computation in the pipeline.

With a naive data placement we will still have problems, e.g. if we have a stride between the elements that is divisible with the number of memory banks. The challenge is to be able to map the programmer's view of logical addresses to different physical ones that avoids the memory bank conflicts.

Figure 1.4: ASIP instruction.

1.4 Method

In order to evaluate a DSP processor we need to know:

Area cost Naturally, the area cost is very important as the intended domain for ePUMA is for applications that are present in high volume products, e.g. smart phones.

Computational performance Is the system capable of fulfilling the performance requirements?

Energy consumption In order to be actually useful in many embedded applications, the energy requirements cannot be too high as many devices operate on batteries.

Programmability The software development cost cannot be higher than that of other capable systems.

Getting an accurate value for the area requires that there is a chip that is fully designed, and that can be synthesized. Also, while it is possible to estimate energy consumption, measurements on a chip running the actual application is the only truly accurate method for power consumption evaluation. It is also necessary to have a chip to measure the actual maximum clock frequency which will influence the computational performance of the device. However, as we can roughly estimate the clock frequency it is possible to use a cycle accurate simulator to get a cycle count when running applications, and this can be used as an indicator of the final performance.

The method used to develop the necessary artifacts for evaluation is the same as described in “Embedded DSP Processor Design: Application Specific Instruction Set Processors” [8].

1.4.1 Evaluation of Software Tools

Measuring the success from a software tools developer point of view is a bit different than evaluating the software for the full system. The development of software tools and hardware platform are of course intertwined, but on the other hand, the software is limited by the capabilities of the hardware. For this, it is useful to define a measure of hardware utilization that can be used to evaluate the quality of the tools. In this project we have opted to define the hardware utilization as

$$\text{Hardware utilization} = \frac{\text{Core instructions}}{\text{Total instructions}}. \quad (1.1)$$

Here, core instructions are the necessary instructions for an algorithm, e.g. the butterfly instructions for an FFT, while the total instructions include control code, calculating addresses etc. This is of course intrinsically linked to computational performance, as it is a function of clock cycles required and operating frequency.

The simplicity of using different software tools, i.e. the programmability of the system, is of course of great importance. This is not something that is easy to measure, and usually involves experiments with large groups of subjects implementing different applications. For this

project, our goal was to be able to program ePUMA without added complexity compared to mainstream systems, by making the sure the tools operate at the same level of abstraction or higher as the tools for other systems. The reason for this is that the primary purpose of ePUMA is to be very efficient, and therefore we will not go further than other systems w.r.t. programmability.

1.4.2 Summary

We can summarize the necessary steps for completing a project as ePUMA as follows:

- Develop the hardware platform.
- Develop software tools such as compilers etc.
- Implement algorithms/applications for measuring the performance of the platform.

These steps are of course performed iteratively and concurrently, to allow for the different tasks to influence each other, i.e. hardware/software co-design.

1.5 Thesis Organization

The thesis is divided into four parts. Part one will provide background information necessary for the remainder of the thesis. Chapter 2 describes the ePUMA platform in more detail. In particular, we will consider the issues with the platform that causes difficulties from a programming point of view. Chapter 3 and Chapter 4 present some solution techniques that are necessary to understand, or at least be convinced that they work, in order to digest the solutions presented in part two.

Part two considers the issues relating to efficient compilation for ePUMA. In Chapter 5 will we consider a model that is suitable for automatic scheduling of tasks and DMA transfers. Chapter 6 describes how we can

optimize our data placement to reduce the cost of data access. In Chapter 7 it is shown how we can compile vector computations efficiently.

Part three describes a programming model based on the results from part two. Chapter 8 goes through how software should be developed on ePUMA and similar architectures based on what we are capable of doing automatically, and what the constraints are for doing so. Chapter 9 describes the various tools that have been developed to help with software development for ePUMA.

Part four summarizes the conclusions and results. Chapter 10 summarizes the results we have gained from benchmarks, while Chapter 11 summarizes the conclusions we may draw from this project, and what might be interesting to look at in the future. Chapter 12 is a retrospective over our process working with this project, what we did right/wrong, and some hopefully useful suggestions to anyone else embarking on projects of this magnitude.

Chapter 2

ePUMA

The ePUMA (embedded Parallel DSP platform with Unique Memory Access) project [3, 9] is a part of the High Performance Embedded Computing (HiPEC) project[10]. ePUMA is an ASIP architecture for DSP processing. In this chapter we will go through the design of the ePUMA architecture and the challenges it poses to the software development tools. However, we will begin with a motivation why a design such as ePUMA is an interesting design to explore.

2.1 Application Characteristics

Given than we wish to target multimedia and baseband processing, i.e. SDR, what features are necessary?

2.1.1 Multimedia

Multimedia applications are characterized by being data parallel and having low temporal locality. There are quite a few properties that are useful to consider when designing an ASIP for this domain:

Parallelism Many algorithms for media are highly parallel, and pretty much any method of parallelization is applicable. For many algorithms the vector lengths of operations are limited when using

vector processing, operations are often performed on blocks of e.g. size 8×8 or 16×16 [11].

Memories Large local memories, i.e. caches or scratch-pads in order to reduce the cost memory of memory access.

Un-aligned data access Many media algorithms require un-aligned data access, and the effect of enabling un-aligned access is quite large, e.g. for MPEG2 as demonstrated in [12].

Pre-fetching/DMA memory transfers The access patterns are often predictable and pre-fetching can be used to speed up the computation. This is one reason why scratch-pad memories can be considered sufficient compared to caches; we still need to issue commands to have the correct data at the right time.

Large data sets One must be able to work on rather large data sets, leading to a large register file or sophisticated addressing. Otherwise a large part of the computation time is spent waiting on load/store instructions.

Application specific instructions The computation can greatly be sped up if common computation graphs are realized as instructions and e.g. only cost one cycle to execute rather than several cycles.

8/16 bit data The number of bits used for media is limited and it is not necessary to use more than 16 bits. This lowers memory size and memory bandwidth requirements.

2.1.2 SDR

Software Defined Radio (SDR) includes programmable symbol processing, programmable parallel bit processing, and programmable Forward Error Correction (FEC) processing [13]. Of these, only the symbol processing part is likely to be mapped to a processor such as ePUMA, while the other two tasks are mapped to ASICs.

SDR is characterized by operations on very wide vectors, and are highly regular in the data access patterns. We can summarize the features needed for an ASIP as follows:

Wide data parallelism The problems for SDR are highly data parallel and can benefit even from very wide SIMD designs, and is the motivation used for e.g. SODA [14].

Pre-fetching/DMA memory transfers The access patterns are very predictable and using pre-fetching is used to speed up the computation. This is one reason why scratch-pad memories can be considered sufficient compared to caches; we still need to issue commands to have the correct data at the right time.

Application specific instructions The computation can greatly be sped up if common computation graphs are realized as instructions and e.g. only cost some cycle to execute rather than several cycles. [15].

8/16 bit data It is also the case for SDR that the values are quite small and it is seldom necessary to use more than 8/16 bit values. [16, 14].

2.2 ePUMA System Overview

The ePUMA system is illustrated in Figure 2.1. The Application Processing Element (APE) controls a set of Compute Clusters (CCs), i.e. it functions as a control core while the clusters are used for the heavy computation. The APE is a simple scalar RISC core. The APE and CCS are connected through a Network-on-Chip (NoC) that includes both a star network (APE to/from CCs) and a ring bus (CC to CC).

The architecture of a CC is shown in Figure 2.2. A CC consists of one Control Processing Element (CPE) and a configurable, at design time, number of Matrix Processing Elements (MPEs). The MPEs are programmable vector processors with addressing features that makes it possible to issue the same vector operation repeatedly with a stride adjustment after each vector operation, enabling the processor to operation on e.g. 2D data

Figure 2.1: ePUMA overview.

sets. As some applications may require custom hardware solutions for some parts, it is also possible to attach accelerators if needed. The Local Vector Memories (LVMs), which is also configurable, are connected to the different computational units and the NoC through a local interconnect network. The allocation of memories to an execution resource is configurable at runtime. For the rest of the thesis, it can be assumed that there are no accelerators, three LVMs and one MPE in the configuration used.

Each LVM consists of N memory banks, as in Figure 2.3. An LVM access consists of a base address and a permutation vector that specifies the offsets from the base address. The address shuffle and permutation blocks send the address to the correct memory bank and to the correct lane respectively. The base address is usually taken from an address register. Address registers are special registers that can be automatically updated with e.g. the width of the data path, or values specified in special step registers. The permutation vector, if not a standard vector access, is fetched from memory. Tables of permutation vectors can be auto-

Figure 2.2: Compute cluster overview.

Figure 2.3: Local vector memory.

matically iterated over, allowing us to access any set of data elements in any order we want without overhead.

2.3 Network on Chip

The cores on ePUMA are connected by a star network as well as a ring network. The star network is for communication between the APE node (and the external DRAM memory) and CC nodes. The ring network is used for communication between different CCs. A detailed motivation for using these kinds of networks can be found in [17]. The computing characteristics for the problems we have looked at in this project falls into two categories:

Data parallel The computation is completely data parallel and a star network is sufficient as the problem is simply to transfer data to the CCs, compute, and transfer the result to the DRAM. This is depicted in Figure 2.4. I.e. each input data block is mapped to an output data block with the same function f .

Figure 2.4: Data parallel computation.

Stream computing The data is processed in different stages, where each stage consumes an input stream and produces an output stream, as illustrated in Figure 2.5, where the functions g and f are performed in sequence, to achieve the function composition $f \circ g$. This

can be achieved on ePUMA connecting the MPEs using the ring network creating a pipeline.

Figure 2.5: Streaming computation.

The majority of this thesis is however, apart from Chapter 5, is mostly concerned with code generation for the MPEs. They differ more from conventional processors and pose greater challenges.

2.3.1 Distributed vs. Shared Memory

Each core has its own memory, and data movement is software controlled through a message passing interface. While this is not that different from e.g. MPI [18], for ePUMA the situation is more complicated due to the fact that the LVMs can be connected to different ports. I.e., when e.g. two LVMs are connected to an MPE, one memory can be accessed from port 0 and the other from port 1. In a way, it can be seen as that the memories can be allocated to different regions in the address space. This can be alleviated by a scheduler as we will discuss in Chapter 5.

2.4 Matrix Processing Elements

The MPE design is the part of ePUMA that is the most different from other systems, and MPE software development will be the focus of the thesis. In this section we will go through the aspects of the MPE design that enables efficient execution of algorithms with predictable data access, and the challenges that it poses to software development.

2.4.1 Application Specific Instructions

In order to accelerate computation it is helpful to have instructions that can perform many arithmetic operations in a single instruction. For example, when computing the Fast Fourier Transform (FFT) it is very good to have a butterfly instruction. When designing an ASIP it is essential to get the set of instructions right, i.e. the instructions should match the application domain well. More specific instructions have a smaller set of algorithms where they can be used effectively, but in turn they can yield better performance. This is very well demonstrated in [15] and [19].

A high number of operations in an instruction, that has dependencies on each other, creates a trade-off between pipeline depth and clock frequency. If a lot of operations are performed in one clock cycle, the clock frequency may suffer due to a longer critical path. If more pipeline stages are added instead, there may be a lot of *nop* instructions executed when we have a dependency between instructions. The effect of a long pipeline may of course be alleviated by branch prediction, but branch prediction results in a more complicated hardware design and larger silicon area. How much is of course dependent on the complexity of the branch prediction strategy.

The design choice for the MPEs is to have quite many stages in the data path, as shown in Figure 2.6. In addition, some of the steps in the pipeline schematic are also internally pipelined. ePUMA is designed for predictable computing, and the drawbacks of a long pipeline without branch prediction is not that severe for these problems, while the benefit of fitting powerful instructions have a large performance impact. One can note that the pipeline is quite regular and almost all parallel steps take the same amount of clock cycles. The exception is the arithmetic units step. It is possible to use long instructions that uses three levels of arithmetic units, and three clock cycles, or only one for short instructions. Short instructions are useful for code with dependencies that also don't use any special instruction that requires several cycles in the execute stage. For any short instruction there is a functionally equivalent long version. One reason is that it may be difficult to schedule instruc-

tions efficiently, as demonstrated in [20], if they are of different length. The version of the MPEs used in [20] had very variable pipeline schedules for different instructions. However, an even larger problem is that it was very difficult to program for, and designing a complete software suit that would handle this was, even if theoretically possible, infeasible with our limited resources.

Figure 2.6: MPE pipeline.

2.4.2 MPEs - Data Level Parallelism using SIMD

As most of the applications that are intended to be executed on the ePUMA platform are data parallel, a SIMD architecture is a quite natural solution. For the MPEs we have taken this one step further allowing the execution of instructions on vectors of arbitrary length. The operands are not required to be in a Register File (RF), but can reside in memory. This is often useful as it is quite common to read a value to the RF, operate on the value, and write the result from the RF to memory, given

that we have specialized instructions that perform many operations at once. This alleviates the need to issue *load/store* instructions, and the actual instructions executed are those necessary for performing the desired computation.

2.4.3 Parallel Memories & Conflict-free Data Access

Given that we have instructions that perform many operations, that memory access is baked into the instruction, and that we have efficient addressing and iteration modes that allow for issuing one core instruction each cycle, we need to have memories capable of delivering vectors equal to the SIMD width. To this end the MPEs have parallel vector memories, as illustrated in Figure 2.3. They are composed by N ($N = 8$ in the current configuration) single-port memories. This enables us to read vectors with addresses of the form

$$\text{addresses} = \text{base address} + \text{offsets}, \quad (2.1)$$

$$\text{offsets} = (e_0, \dots, e_m), \quad (2.2)$$

$$m < N. \quad (2.3)$$

However, as a consequence of using single-port memories, in order to be able to actually fetch a vector all addresses in an access need to reside in different memory banks. That is, if

$$\text{addresses} = (a_0, \dots, a_m) \quad (2.4)$$

then

$$M = \{0, \dots, m\}, \quad (2.5)$$

$$\forall \{(i, j) \in M \times M \mid i \neq j\} : a_i \not\equiv a_j \pmod{N}. \quad (2.6)$$

Chapter 6 will show how we can exploit these properties to in order to reduce the cost of memory access by relocating the physical addresses of data elements.

SIMD vs. Superscalar

A superscalar processor [21] is very pleasant from a software development point of view, as it will try to figure out how execute instructions in parallel at runtime, and the programmer gets a speedup without effort. Designing a superscalar processor is on the other hand tricky and requires more hardware and power per functional unit [22]. With predictable problems this is not necessary, as the scheduling can be performed offline. Therefore, it is a bit luxurious for a low-power/area design. Of course, there are superscalars with SIMD capabilities, as e.g. Intel processors with SSE3.

SIMD vs. VLIW

VLIW architectures (e.g. [23] or [24]) are a bit more suitable for general code compared to the MPEs. I.e., it is possible to schedule the functional and load/store units to work in tandem for arbitrary code, while a SIMD architecture is locked by the instruction set. On the other hand, compiling efficient code for VLIW is not trivial. For example, attempting to compile an FFT as in [25] and [26] yielded very low performance, i.e. an order of magnitude lower performance, compared with the hand optimized version. While there are efficient manually created implementations, they are very complicated, and the MPE version is much simpler.

2.5 Architecture comparison

There are quite a few architectures that share some similarities with ePUMA and are intended to be used for the same domains. In this section we will consider the similarities, but more importantly differentiate what is different about ePUMA, and motivate why the ePUMA approach is interesting to investigate.

2.5.1 Imagine

The Imagine stream processor [27, 28] is a media processor with a quite interesting design. First of all, it has a 128 kB streaming register file that contains streams that eight VLIW clusters execute stream kernels on in a SIMD style manner, i.e. the same VLIW instruction is issued to all eight clusters. A kernel program is a sequence of VLIW instructions that can operate on elements in the stream. Each VLIW cluster contains a quite large register file as well, 256×32 -bit.

One major similarity between Imagine and ePUMA is that the MPEs can issue vector operations on arbitrary length, similar to using a kernel on a stream. In the MPE case, this is of course only one instruction, while for Imagine it could be an arbitrary program. On a higher level though, i.e. if we compare each MPE with one cluster, the MPEs are more general than the clusters, in that they may run independent code. The overall programming of a system full system is in a way similar as the CAL language [29]. CAL is the intended high-level language for the HiPEC project [10], and is a dataflow language. While CAL is more general than the environment for Imagine it is unlikely to matter for the problem domain.

2.5.2 VIRAM

VIRAM [30, 31, 32] is a media processor with a SIMD width of eight, the same as for the MPEs. It has a scalar processor and vector co-processor. It is a quite conventional processor if not for the ability to not only access data in a sequential way, but also with indexed load, i.e. load data from memory based on a vector register (similar to our permutation vectors), and strided access. A major difference is there is a system that can handle latencies and can run many accesses in parallel, and will get high performance as long as the throughput is high. The non-sequential are on the other hand slower than for the sequential case, which is not true for ePUMA unless there are bank conflicts.

2.5.3 CGRA

Another architecture within the HiPEC project is the Coarse-Grained Reconfigurable Array [33], designed for algorithms with a low computing buffer and long pipeline. The overall architecture consists of some configuration of tiles that send data between each other. E.g. a scalar tile for scalar computations, a vector computation tile and memory tiles. The implementation in [33] is quite similar to a compute cluster, except that memory access involves several tiles and the addressing and data access does not share a pipeline with the computation. Also, the CGRA is micro-programmable and the instructions are not fixed and can be configured at runtime, whereas in ePUMA the instructions are only configurable at design time. Our approach saves area but loses generality.

2.5.4 SODA/Ardbeg/AnySP

SODA (Signal-processing On-demand Architecture) [14] is an SDR processor and uses an ARM as a master processor and four processing elements. Similarly to the compute clusters the processing elements use scratch-pad memories, and has a scalar and SIMD path. The AGUs are, in contrast to the MPEs, in a separate pipeline. The SIMD path is 32 words wide and has a shuffle network that sits before the ALUs and multipliers. The access to the SIMD scratch-pad memory is between the shuffle network and execution stage, so while it is possible to perform some arithmetic operations while reading data, it is still necessary for the data to end up in the register file.

Ardbeg [34] is a continuation of SODA where the SIMD ALUs and multiplier units have shuffling networks included. The scalar part has also merged with the AGU. The SIMD width is kept at 32 based on a computation delay and energy efficiency compared with area.

AnySP [11] is another evolution of SODA where the design is intended to work for media applications as well. The thing of main interest of this design is the choice of variable SIMD width. The reason is the poor utilization of the SIMD units as while there is an abundant

amount of parallelism for media applications, the SIMD operations are of limited width.

2.5.5 EVP

EVP (Embedded Vector Processor) [16] is a processor for SDR. It is a VLIW features 16 words wide SIMD instructions (though configurable), and can issue five instructions at a time in addition to four scalar operations, and also update address registers and handle loop control. EVP uses a register file (16×16) and a shuffling unit (that runs in parallel with computation) to support reordering of data.

EVP is programmed using EVP-C, a superset of C, where the superset of vector types and intrinsics are compiled using vector instructions while the rest is compiled using scalar instructions.

It seems that they can compile code quite efficiently for it, e.g. a 64 point FFT requires 79 cycles compared to 64 for manually scheduled code, i.e. an overhead on only 23.4% [16]. The manually scheduled code utilizes the multipliers during 48 of the 64 cycles. That is, the utilization is 75%. However, it is unclear how the results for FFTs that will not fit into their register file will be affected.

What separates the MPEs from EVP is that the addressing is in the same pipeline as the vector processing unit. It shares the scratch-pad memory design.

2.5.6 BBP

The BBP processor [19] is designed specifically for SDR. It is conceptually similar to a compute cluster in that there is a control processor, and scratch-pad memories that are operated on by vector execution units, i.e. a DSP controller that issues vector instructions. These units are less general than the MPEs and are tailored for a narrower domain. However, the commercial success of this design shows that the assumptions about the SDR domain are valid and that a more general design should not have any problems, unless the power and area requirements become too high.

2.6 Summary

There are quite a few design choices that ePUMA shares with other architectures designed for the same domain. ePUMA uses scratch-pad memories for the processing elements, can issue wide SIMD instructions and so on.

In contrast to most other architectures, the different hardware blocks for e.g. addressing, permutation, computation etc. are organized in a long pipeline, i.e. we utilize pipeline parallelism instead of using a VLIW approach with several pipelines. This may make ePUMA less suited for general computation, as code with many dependencies would execute slowly. However, the compute clusters still have a scalar core for such code, and often such problems come in bulk and can be parallelized over problems. E.g. if we consider operations on 2×2 and 4×4 matrices for LTE downlink, the amount of matrices is either 128, 256, 512, 1024, 1536, or 2048 (that is the amount of sub-carriers supported by LTE) that can be processed in parallel, and there will be an abundance of data parallelism.

Compared to the other architectures, the MPEs have many more stages in the execution stage of the instructions. This allows for more powerful application specific instructions that reduces the need to use a register file, or in other words, the MPEs performance rely on good instruction selection.

Chapter 3

Optimization

In order to solve some of the issues w.r.t. to automating tasks in the software development flow for ePUMA the problems have been formulated as mathematical optimization problems. In this chapter we will briefly go through some ways to formulate problems that can be solved with standard solvers. The purpose is not give an explanation of how these solvers actually work; however, an informal motivation as to why they do work will be provided.

3.1 Linear Programming

For linear programs, i.e. a mathematical formulation of an optimization problem that we can solve using standard solvers, the objective function is required to be linear. In addition, the constraints must also be linear. All linear programs can be written according in *standard form* as in Equation (3.1)

$$\min c^T x, \text{ subject to } Ax = b, x \geq 0. \quad (3.1)$$

However, it is possible to formulate using either min or max, and use $\leq$ and $\geq$ in addition to $=$. Here, c and x are vectors in $\mathbb{R}^n$, b is a vector in $\mathbb{R}^m$ and A is an $m \times n$ matrix. These problems can be solved with either the *simplex* or *interior-point* methods, which both are described in [35]. Here we assume that the reader has basic knowledge of the simplex method

and we will not explain how the linear programs are solved when describing (Mixed) Integer Linear Programming, (M)ILP.

3.2 (Mixed) Integer Linear Programming

An integer linear program is similar to the linear case, except that x is a vector of integers ($x \in \mathbb{Z}^n$). For Mixed Integer Linear Programming (MILP) only a subset of the elements in x are constrained to be integers, while the others are non-integers. Integer Linear Programming (ILP) problems are very computationally demanding to solve in the general case, as they are NP-complete.

A comprehensive introduction to MILP can be found in e.g. “Integer and Combinatorial Optimization” [36].

3.2.1 Exact Solution Strategies

If a problem is defined as an integer linear program, it can be solved exactly (i.e. finding the global optimum) using any of the following methods.

Branch and Bound

Relax the integer constraints so that we get an LP-program. Solve the program and see if any variable that should have an integer value has a non-integer solution. E.g. as for X in Figure 3.1, where the nodes represent the values of the optimal solution for the relaxed problem. By creating two new programs, one with the constraints $X \leq 2$, and another one with $X \geq 3$, we can recursively solve them and return whichever returns the most optimal value.

Cutting Plane

Another way of continuing after finding a non-integer solution is to cut the current optimal solution using an additional constraint. Given a convex problem we know that the solution will always be in a corner, and

Figure 3.1: Example of branch and bound.

there will always exist a cut that removes the current optimum while not removing any feasible integer solution. Exactly how these constraints are found will not be explained here. For example, say that we wish to optimize

$$\max x + y \quad (3.2)$$

when x and y are integers with the following additional constraints, where we denote Equation (3.3) as constraint 1 and Equation (3.4) as constraint 2.

$$2 \cdot y \leq 7 \quad (3.3)$$

$$4 \cdot x + y \leq 19 \quad (3.4)$$

$$x \geq 0 \quad (3.5)$$

$$y \geq 0 \quad (3.6)$$

These constraints are shown in Figure 3.2. Solving this as an LP problem gives us an optimal solution of 7.375 with $x = 3.875$ and $y = 3.5$. We can then add a cut as in Figure 3.3, i.e. add a constraint which in this case is

$$x + y \leq 7, \quad (3.7)$$

Figure 3.2: Example cutting an invalid optimum.

to remove this solution, and try again. With this cut we get a solution with optimal value 7 with $x = 3.5$ and $y = 3.5$. Cutting this solution with

$$x + 2 \cdot y \leq 10, \tag{3.8}$$

as in Figure 3.4 gives us an optimal solution of 7 with $x = 4$ and $y = 3$, which is a valid solution. In this case the solver could potentially have given an acceptable solution before the last cut, as the objective function value was the same.

Branch and cut

Branch and cut is a combination of using both branch and bound and cutting planes.

3.2.2 Benefits

There are many benefits to using an ILP approach.

Figure 3.3: Example of cutting an invalid optimum.

- There are many available solvers of excellent quality for MILP. E.g. Gurobi [37] or Google or-tools [38].
- It does not require any specialized algorithm to be developed.

3.2.3 Drawbacks

Unfortunately, there are also some drawbacks that might force us to use other approaches.

- Can be a lot slower for problems that has a structure that allows specific algorithms to be developed for it.
- Can sometimes be hard to describe the problem.

Figure 3.4: Example of cutting an invalid optimum.

3.3 Constraint Programming

Another way to solve similar problems is to formulate Constraint Satisfaction Problems (CSPs) instead. A more complete introduction to the area than will be provided here can be found in e.g. “Constraint Logic Programming using Eclipse” [39].

3.3.1 Definition

A CSP is a 3-tuple with $(\mathbb{V}, \mathbb{D}, \mathbb{C})$. $\mathbb{V}$ is a set of variables, where each variable is associated with a domain in $\mathbb{D}$. A constraint in $\mathbb{C}$ specifies what combinations of values that are allowed.

3.3.2 Constraint Propagation

One method of solving these problems quite efficiently is by constraint propagation. Essentially, when one variable is assigned a value, we then

check that the solution is still consistent, i.e. that no constraint is violated. We reduce the possible values for other variables and as long as a variable doesn't end up with an empty domain, we can continue with assigning another variable a value. When a domain is empty, we need to backtrack and try some other value.

3.3.3 Example

Assume that we have the variables (with associated domains) $x \in \{0, \dots, 4\}$, $y \in \{1, \dots, 4\}$, and $z \in \{0, \dots, 3\}$ and have one constraint

$$x > y + z. \quad (3.9)$$

We begin with the state as in Figure 3.5a. Each box contains a tuple with the level, i.e. how deep we are into the search and serves as a marker for how far to backtrack if we reach inconsistent states, or search for more solutions, and the remaining values of the domain to try.

If we set $x = 0$, we can remove that value from level 0, and add another level of possible values as in Figure 3.5b. However, there are no values of y and z that are consistent with the constraint, and need to backtrack to level 0 (Figure 3.5c).

However, if we would try $x = 3$ we would still have non-empty domains for y and z as in Figure 3.5d. Here, the domains are constrained to $\{1, 2\}$ and $\{0, 1\}$, as

$$y_{max} + z_{min} \leq 2, \text{ and} \quad (3.10)$$

$$y_{min} + z_{max} \leq 2 \quad (3.11)$$

must hold.

Letting $y = 1$ gives us Figure 3.5e. This does not change the domain for z and it remains as it is. Letting $z = 0$ gives an acceptable solution as in Figure 3.5f. I.e., $x = 3$, $y = 1$, and $z = 0$ satisfies the constraint. At this point we could backtrack to search for more solutions.

Figure 3.5: Constraint solving process.

3.3.4 Labelling

In Section 3.3.3 the selection order of values to try was not consecutive. In general, the method used can usually be configured in most solvers, e.g. whether to start with the variable with the smallest/largest domain, small values before large or vice versa, or even starting with values in the middle. The selection method used can influence the solution time quite significantly for some problems, so it can be useful to consider which values would be more likely to be acceptable.

3.3.5 Optimization

While the definition of CP given in Section 3.3.1 does not mention anything about optimization most solvers allow adding an objective function. A trivial way of adding optimization is to add the objective function as a constraint and resolve the problem with tighter and tighter bounds until the problem is infeasible. However, most good implementations are likely a bit more clever.

3.3.6 Useful Constraints

The biggest benefit of CP is that there are many standard constraints that makes it very easy to model problems. For example, the *alldifferent* constraint that ensures that a set of variables have different values, can be useful for assigning memory banks to addresses. Another constraint is *cumulative* which is very useful when scheduling, as it ensures that a set of tasks that uses a set of resources do not overlap in time.

Besides just making the modelling easier, CP implementations can provide optimized algorithms for the problem and improve the execution time considerably as it does not need to use the more general search procedure of ILP. Unfortunately, combining different constraints in the wrong way may lead to very poor performance.

3.3.7 Benefits

As with ILP, there are quite a few properties of CP that makes it interesting to use a generic tool for solving problems.

- As for (M)ILP, there are many good solvers, e.g. JaCoP [40] or Gecode [41].
- It is easy to apply own heuristics which is useful if one has an idea about what the solution will look like.
- If it is possible to use standard constraints it offers great solution times.
- The work required to create a correct model is much easier than for ILP.

3.3.8 Drawbacks

The major issue with CP is that if the specialized algorithms cannot be applied or interact badly the performance can be horrible.

3.4 Other methods

In the ePUMA project CP has mostly been used for prototyping and have later been replaced by ILP for the final implementations of different tools. This is primarily due to the fact that the performance of the ILP solvers have been significantly better to warrant the extra implementation time, and that these solution times have been reasonable enough for our problem sizes. However, should it be necessary to solve the problems formulated later in this thesis on problems that makes the solutions times infeasible, there are other approaches, called meta-heuristics, that do not necessarily find the most optimal solution, but often finds reasonably good solutions in a reasonable time frame.

A meta-heuristic is a method that can be applied when we either don't know what the optimal solution looks like, i.e. determine whether

something is optimal or not, and/or do not know how to search for it. All that is required is that it is possible to evaluate an objective function, and the search for better results is done in a probabilistic manner. These methods do often produce optimal or near-optimal results [42, 43], which makes them interesting to use for otherwise intractable problems. However, which heuristic gives the best results are very problem specific [44], and trying several methods is recommended.

3.5 Summary

In this chapter a brief overview of ILP and CP has been provided as well a short introduction to how these problems are solved. We will later use these methods as general techniques for solving problems, as all that is needed is a formulation of what should be true for a valid solution. Given enough time these method will find one, and even the most optimal one¹, if the problem has a feasible solution.

¹With respect to the model.

Chapter 4

Dataflow analysis

In this chapter we will briefly go through the basics of dataflow analysis. The purpose is to motivate the reasoning of using the equations in Chapter 7, and why they will work, rather than a full formal treatment or getting into details on how to optimize the procedure.

4.1 Purpose

The purpose of dataflow analysis is to gather information about a program in order to be able to optimize it and/or detect faults. E.g. if we can conclude that the result of a statement is never used, then we can safely remove it from the program and consequently execute the program using fewer instructions¹, or if we can deduce that a statement e.g. uses an uninitialized variable we know that it is a faulty program.

4.2 Example program

Assume that we have a program that is represented by quadruples as in Listing 1. The term quadruples is often used because most statements

¹Technically, the cost of issuing the instruction could be completely hidden by data access and/or affected by a myriad of other hardware issues. However, in general performing less work translates to at least some reduction in execution time.

are of the form

$$a \leftarrow b \odot c. \quad (4.1)$$

I.e., two sources, one destination and one operation.

Algorithm 1 Sample function

```

1: function EX( $A, I$ )
2:    $i \leftarrow 0$ 
3:    $r \leftarrow 0$ 
4: Loop:
5:   if  $i \geq I$  then goto EndLoop
6:    $i \leftarrow i + 1$ 
7:    $r \leftarrow r \odot A[i]$ 
8:   goto loop
9: EndLoop:
10:  return  $r$ 

```

As a simple example we can use *reaching definitions* that is quite useful for many optimizations. I.e., if we take e.g. the definition of i on line 2, which statements *may* be influenced by this definition? We know that if the statement on line n is executed the definition d_n is a valid definition for coming statements. At the same time, if a variable t is set all other definitions are invalid. The definitions that affect the statement on line n can be defined as

$$in_n = \bigcup_{p \in pred_n} out_p \quad (4.2)$$

out_p is the valid definitions after statement p , and naturally the valid definitions are the union of all predecessors, $pred_n$, as we are looking for any definition that *may* be valid while executing statement n . The valid definitions after statement n is then

$$out_n = \{d_n\} \cup (in_n - def_{st}), \quad (4.3)$$

where all definitions of the variable t , def_{st} , are invalidated (i.e. removed from the set) and the new definition of t , d_n , is added to the set.

In general, we have that

$$in_n = join(\{out_p \mid p \in pred_n\}) \quad (4.4)$$

and that

$$out_n = tf(in_n) = join(\{out_p \mid p \in pred_n\}). \quad (4.5)$$

Assuming that the set of objects can be ordered into a partially ordered set, with a finite height (or bottom), if the function composition $tf \circ join$ is monotonically increasing (or decreasing) we are guaranteed termination [45].

Calculating the reaching definitions for Listing 1 is performed iteratively. First, we have the predecessors of each statement in Table 4.1. We

Table 4.1: Predecessors

n	Predecessors
2	None
3	2
5	3 and 8
6	5
7	6
8	7
10	5

will update the states of each statement in the same order as they appear in the table.

In Table 4.2 we have the states for the iterations when calculating the reaching definitions. Being the first statement, statement 2's input is always the empty set. Statement 3 will always have the output from statement 2 as input, which will always be d_2 . Statement 3 adds d_3 and passes it on to statements 5 and 6, where statement six replaces d_2 with d_6 . As we haven't calculated a value for statement 8 yet statement 5 uses the empty set for the merge. Statement 7 replaces d_3 with d_7 , statement 8

passes the same values, and statement 10 takes the output from statement 5.

For the second iteration, a change occur for statement 5 as statement 8 has $\{d_6, d_7\}$ as output, making the output for statement 5 $\{d_2, d_3, d_6, d_7\}$. Statement 6 and 7 removes d_2 and d_3 before we reach statement 8. Statement 10 is updated with the new output from statement 5.

Running the same procedure again results in the sets shown for iteration 3, shown in Table 4.3. As this is identical to the sets for iteration 2, we have found a fix-point and terminate.

Table 4.2: Iterations

n	Iter. 1		Iter. 2	
	in_n	out_n	in_n	out_n
2	$\emptyset$	$\{d_2\}$	$\emptyset$	$\{d_2\}$
3	$\{d_2\}$	$\{d_2, d_3\}$	$\{d_2\}$	$\{d_2, d_3\}$
5	$\{d_2, d_3\}$	$\{d_2, d_3\}$	$\{d_2, d_3, d_6, d_7\}$	$\{d_2, d_3, d_6, d_7\}$
6	$\{d_2, d_3\}$	$\{d_3, d_6\}$	$\{d_2, d_3, d_6, d_7\}$	$\{d_3, d_6, d_7\}$
7	$\{d_3, d_6\}$	$\{d_6, d_7\}$	$\{d_3, d_6, d_7\}$	$\{d_6, d_7\}$
8	$\{d_6, d_7\}$	$\{d_6, d_7\}$	$\{d_6, d_7\}$	$\{d_6, d_7\}$
10	$\{d_2, d_3\}$	$\{d_2, d_3\}$	$\{d_2, d_3, d_6, d_7\}$	$\{d_2, d_3, d_6, d_7\}$

In practice, transfer functions are generally defined over basic blocks rather than individual statements for performance reasons. Also, updating each basic block one after another is rather wasteful. By using the worklist approach, we initially add each root to a worklist. After a block is processed, its successors is added to the worklist. As the state for a block will only change if the input is changed, we will not needlessly update the output values.

Table 4.3: Iterations

n	Iter. 3	
	in_n	out_n
2	$\emptyset$	$\{d_2\}$
3	$\{d_2\}$	$\{d_2, d_3\}$
5	$\{d_2, d_3, d_6, d_7\}$	$\{d_2, d_3, d_6, d_7\}$
6	$\{d_2, d_3, d_6, d_7\}$	$\{d_3, d_6, d_7\}$
7	$\{d_3, d_6, d_7\}$	$\{d_6, d_7\}$
8	$\{d_6, d_7\}$	$\{d_6, d_7\}$
10	$\{d_2, d_3, d_6, d_7\}$	$\{d_2, d_3, d_6, d_7\}$

4.3 Summary

This chapter provided a brief explanation of dataflow analysis with as simple example. If necessary, more detailed explanations can be found in [46], [45], and [47].

Part II

Compilation Techniques

Chapter 5

Scheduling

Writing code that utilizes multi-cores efficiently is often difficult and tedious work. This is especially true for ePUMA as we also need to keep track of which memories we're currently using for computation and data transfer. This section describes how we can optimize the schedules of different tasks necessary to implement a kernel.

Both ILP and CP have been used for scheduling before (e.g. [48, 49] and [50]). While it is quite easy to formulate a model that provides good schedules when solved, a scheduler should provide fairly good schedules quite rapidly during the development phase, while it might be acceptable for longer solution times for a release build.

The point of creating a scheduler for ePUMA can be summarized as follows:

1. Is it possible to create a scheduler for ePUMA so that the complexity created by all memories, especially the non- static memory mapping (i.e. LVMs), are hidden?
2. Will it deliver schedules as good as or better than schedules created by a human? For DSP inefficient solutions are not acceptable.

5.1 Problem/Model Definition

The problem that the scheduling method described here is used to solve is to map a task graph onto the different compute clusters. We require this to be a *directed acyclic graph*. For our purposes this is not that much of a restriction, as most kernels we are considering can be unrolled etc. to have this form.

Our basic unit for scheduling will be a task. A task is defined as something that takes inputs and outputs, specified by memory port and size, and has an execution time. In other words a task t is a function that maps data residing in memories connected to some ports, and places the output in memories connected to some ports, as in Equation (5.1) where the subscripts denote memory ports.

$$t : I_k \times \dots \times I_l \rightarrow O_m \times \dots \times O_n \quad (5.1)$$

A graph of tasks is constructed that are connected by inputs and outputs.

5.1.1 Events

In this model each activity and data blocks will be associated with two events, *begin* and *end*. These events will contain when something begins or is created, and when something ends or is deallocated. They are modelled as continuous variables that marks the time from the first event.

5.1.2 Resources

Finding a correct schedule requires that the resources in ePUMA are not used simultaneously by different tasks. The resources we are concerned with in this model is the clusters, the memories, and the links for data transfer.

When modelling the problem we can view all resources as a set $\mathbb{R}$, and that each activity to be scheduled requires a subset of these resources. Then it is simple to add constraints that if any resource is shared, the activities cannot overlap in time. Whether or not a resource is used by

an activity, is determined by a set of possible resource decision variables that can assume the values 0 and 1, one indicating that the resource is used.

5.1.3 Activities

There are two types of activities that we will schedule:

Call When the actual computation of a task occurs.

Transfer A transfer of a data block between memories.

We will denote the set of all activities as $\mathbb{A}$.

To make sure that no resource conflicts occur, if two activities requires resources that could potentially overlap we add the following constraint:

$$\forall r, a, a' \in \mathbb{R} \times \mathbb{A} \times \mathbb{A} : \text{overlap}(a, a') + a_r + a'_r \leq 2. \quad (5.2)$$

overlap is an expression that becomes one of the activities overlap in time, and $a_r \in \{0, 1\}$ means that resource a uses resource r . I.e. if both activities uses a resource the overlap must be zero.

An overlap constraint can be formulated in the following way. If we have a binary variable *smaller*, and two variables a and b that are integer variables. Then we can add the constraints,

$$a - b < M0 - M0 \cdot \text{smaller}, \quad (5.3)$$

and

$$a - b > -M0 + M0 \cdot \text{smaller} \quad (5.4)$$

Here $M0$ is a large constant that is sufficiently large. If *smaller* is one, then Equation (5.3) will be the active constraint. Rewriting Equation (5.3) assuming *smaller* = 1 we get

$$a - b < M0 - M0 = 0, \quad (5.5)$$

i.e. $a < b$. With *smaller* = 0 we get

$$a - b < M0, \quad (5.6)$$

that will be true given a large enough value for $M0$, and the constraint is inactive. The same kind of reasoning can be used for Equation (5.4), forcing $a > b$ when $smaller = 0$. Using this, it is simple to adjust the equation to adjust for intervals. E.g.,

$$a_e - a'_b < M0 - M0 \cdot smaller, \quad (5.7)$$

and

$$a_b - a'_e > -M0 + M0 \cdot smaller. \quad (5.8)$$

Here, subscript b denotes the start time, and e the end time.

5.1.4 Data blocks

Data blocks are the inputs and outputs of tasks. They are allocated to some clusters' memory or the main memory. A data block is created by either a call, if an output data block, or a data transfer, for an input data block.

Input data blocks must have been created before a task is executed, and they are created by a transfer.

$$\forall b \in \text{input blocks} : b^{begin} > transfer_b^{end} \quad (5.9)$$

After all blocks for a task has been created, the call may be executed.

$$\forall b \in \text{input blocks} : b^{begin} < call^{begin} \quad (5.10)$$

Even though it may seem that this (i.e. always needing to transfer data blocks before using them as input data), would cause unnecessary movements of data, there is no actual overhead. If data should be moved to another core, the movement cost is unavoidable. Within a core, we still have to set the address registers, and the transfer cost will then be zero. We consider all input blocks to be alive until at least until the call is finished, i.e.

$$\forall b \in \text{input blocks} : b^{end} > call^{end}. \quad (5.11)$$

For output blocks, they are considered to be created after the call has finished,

$$\forall b \in \text{output blocks} : b^{begin} > call^{end}, \quad (5.12)$$

and may then be transferred to some other memory,

$$\forall b \in \text{output blocks} : b^{begin} < transfer^{begin}. \quad (5.13)$$

They must remain alive until the transfer that moves it has completed,

$$\forall b \in \text{output blocks} : b^{end} > transfer_b^{end}. \quad (5.14)$$

A data block is allocated to one and only one memory. We formulate this as

$$\sum_{m \in \mathbb{R}_{mem}} b^m = 1, \quad (5.15)$$

where $b^m = 1$ if the block is allocated to memory m .

5.1.5 Transfers

A transfer can either be point-to-point or broadcast. A broadcast may occur if data in main memory have several destinations. In this case a transfer has one input block and several output blocks.

The execution time of a transfer is determined by a setup time and the actual time to transfer the data depending on the block size.

$$\forall t \in \text{transfers} : t^{end} = t^{begin} + t^{setup} + t^{size} \quad (5.16)$$

The cost for a transfer may be zero if the source and destination blocks are located in the same memory. E.g., if we have a decision variable dm that is one if the blocks are placed in different memories and the block size bs for the blocks b_i and b_o , we may define

$$t^{size} = dm \cdot bs. \quad (5.17)$$

Each block is placed in one memory, so if we add a constraint for each memory,

$$\forall m \in \mathbb{R}_{mem} : dm \geq b_i^m - b_o^m, \quad (5.18)$$

dm will be forced to one if the blocks are placed in different memory banks. The same method can be used for t^{setup} .

The memories used by a transfer are the same as for the related data blocks. I.e.

$$\forall r, b \in \mathbb{R}_{mem} \times \text{related blocks} : transfer_r \geq b_r. \quad (5.19)$$

The links used are determined by

$$\forall m, b \in \mathbb{R}_{mem} \times \text{related output blocks} : \quad (5.20)$$

$$\sum_{l \in preds(m)} transfer_l = b_m - equal(b_m, input_block_m)$$

and

$$\forall m, l \in \mathbb{R}_{mem} \times \mathbb{R}_{links} : \quad (5.21)$$

$$\sum_{l' \in preds(l)} transfer_{l'} + input_block_m \geq transfer_l$$

That is, if a block b is allocated to the memory resource m , denoted by b_m , and the input block is allocated to memory attached to the same core, no link connected to that node should be active. Otherwise, exactly one link should be active (Equation (5.20)). If a link is used, some predecessors link must be used, or the source block should come from the node itself (Equation (5.21)).

5.1.6 Calls

All calls must be mapped to a processing element. Currently the only option is to map calls onto a compute cluster, but could be anything e.g. an accelerator.

$$\sum_{r \in \text{valid processing elements}} call_r = 1 \quad (5.22)$$

The input and output blocks must also be mapped to memories connected to the processing element.

$$\forall pe, db \in \text{valid processing elements} \times \text{data blocks} : \quad (5.23)$$

$$call_{pe} - \sum_{m \in memories(pe)} db_m = 0$$

I.e., if a processing element is chosen we must force the data blocks to be mapped to related memories.

The events for all calls are constrained by the following equation.

$$\forall c \in \text{calls} : c^{end} = c^{begin} + c^{time} \quad (5.24)$$

5.1.7 Objective Function

The objective function f is simply defined as the end of the latest activity.

$$\forall a \in \mathbb{A} : a^{end} \leq f \quad (5.25)$$

5.1.8 Partitioning and Symmetry Breaking

Implementing the model described straight off leads to some performance problems. First off, it is unnecessary to add resource checking constraints between all activities as in Equation (5.2). By looking at the graph we can add simply ignore the resource constraints for activities that have dependencies between them.

Secondly, for data parallel problems we will have many symmetrical solutions. This can be solved quite easily by splitting the tasks into sets and creating artificial dependencies between the sets. While this may create less optimal results than the original model, i.e. a set size of one would obviously be detrimental, some smaller multiple of the number of cores depending on the communication/processing ratio seems to be sufficient.

This problem can be further alleviated by simply stating that if we for example have M processing elements and two tasks with index i and j , and $i \equiv j \pmod{M}$ holds, then they should be mapped to the same element. A constraint that if $i \equiv m \pmod{M}$, then task i will be mapped to element m will reduce the symmetries even further without degrading performance. The performance might be affected if we try to schedule a data parallel section before/after a non-parallel section.

Of course, applying constraints can simply be controlled by a switch to the scheduler that determines the optimization level.

5.2 Conclusions

Unfortunately this scheduling method has not been tested on any real problems. While it has performed well on synthetic problems and generating equivalent schedules of some hand implemented kernels, the lack of integration has been a problem for wider use. The APE and CCs has been constantly changing during the project as well as the interfaces, which would make the effort to create and maintain a code generator too large to be of real value, at least w.r.t. the questions we wanted answered.

However, we can conclude that:

1. An automatic scheduler is certainly possible, and by its construction we know that it will generate good schedules, at least if the time used for creating the schedule is of no concern. For input graphs with a shape similar to the ones occurring in the intended application domain the solution times are reasonable unless for large data parallel problems with many symmetrical solutions.
2. By applying some constraints mimicking the decision process of a human, the symmetries and possible solution space are reduced and will be solved near instantly with as good quality as a human. Admittedly, to make this point really convincing the number of test cases would need to be much larger and used on real world problems, but the engineering effort to make this point was not possible in the required time frame.
3. The results are not really surprising, and it is possible to make a fairly good argument as to why the ePUMA system design will not negatively affect programming productivity for the intended application domain.

Chapter 6

Conflict-free Memory Access

In this chapter we will consider different methods for dealing with the issues of parallel data access on the MPE cores. That is, how we can ensure that we will be able to utilize the full memory bandwidth available to us.

6.1 Motivation

Data access is often the reason for low performance in many systems. In extreme cases, the efficiency can be as low as 1-12% [51]. These figures do include the full memory system of a general purpose processor. The metric used is

$$\frac{\text{True/core instructions}}{\text{total cycles}}, \quad (6.1)$$

i.e. the fraction of clock cycles necessary if it would be possible to use the hardware in an ideal way. One can note that much of the overhead for quite a few kernels that are common in DSP, as surveyed in [52], can be removed by adding hardware looping, address generation, and address transformations [51]. One reason for poor efficiency is that data require shuffling. For example, in [53] they present an algorithm for optimizing the number of shuffling instructions for SIMD computation. Even

though they manage to decrease the number of shuffling instructions immensely, they still have many left. In Table 6.1, we can see the improvement. Here, "Base" is a baseline algorithm for generating shuffling instructions, "Opt" represents the most optimized version, and "Mis" represent the case when the data is misaligned. We will return to these

Table 6.1: Number of shuffling instructions.

		VMX			SSE2		
Algorithm	Size	Base	Opt.	Mis. Opt.	Base	Opt.	Mis. Opt.
FFT	16	96	24		96	24	
FFT	32	208	48	64	208	48	64
FFT	64	352	96		352	96	
Bitonic	16	52	34		56	34	
Bitonic	32	136	92	100	144	92	100
Bitonic	64	336	232		352	232	

values in Chapter 10. However impressive the reduction is, it will still constitute a large portion of the execution time for an ASIP with specialized butterfly instructions.

In [54] a method for generating shuffle instructions for permuting data that will need the number of cycles in Table 6.2 to perform the permutation. For stride of 2 and a size of 4 means that it takes 4 cycles for the Core2 to permute (x_0, x_1, x_2, x_3) into (x_0, x_2, x_1, x_3) . We will see in Chapter 10 why these values are also insufficient.

Table 6.2: Number of shuffling instructions.

Stride	Size	Cycles for Core2 SSE2	Cycles for Cell SPU
2	4	4	15
4	16	13	22
8	64	35	
16	256	106	

6.2 Related Work

There’s been a lot of work in the field of data permutation. The work in [55] is very similar in that it provides a method for generating hardware for arbitrary permutations in streaming applications. However, it uses two RAMs and the process is to accept the input vectors into one of them, then transfer the data in the first RAM to the second in some order while permuting the data, and permit the output vectors to be accessed from the second RAM. This middle step isn’t really available in the MPEs, unless one issues instructions that does the same, which would add at least

$$\left\lceil \frac{\text{buffer size}}{\text{SIMD width}} \right\rceil \tag{6.2}$$

clock cycles to the execution time. By adding some assumptions about the access patterns in [56] this drawback is removed. It does however turn out that the assumptions made are unnecessary, since it is always possible to achieve conflict-free access as long as each value is written and read only once [57]. We will revisit this later in this chapter.

The idea of permutation have been used before on register files [58, 59]. By using permutation on the register file access of non-consecutive elements is possible. However, the register file is quite large. The approach in [58] is quite similar to the approach for the eLite DSP architecture presented in [60], where the register file can be addressed by vector pointers.

SIF [61] is a quite interesting solution. A Permutation Vector Register

File (PVRF) is used, where a permutation is implicitly applied on a vector register when accessing the register. This is controlled by setting state registers for each vector register. This is quite unobtrusive w.r.t. the rest of the system and is a potential solution for any system. However, the values available when permuting data is still limited to a single vector register.

6.2.1 Why Multi-bank Memories?

Many other architectures have had great success with multi-bank memories. For example, in [12] the effect of using single-bank, multi-bank and multi-port memories when implementing an MPEG2 encoder is demonstrated. Comparing the systems with the different memory configurations gave a speedup up of 2.24 when using multi-bank memories compared to a single-bank memory, and a speedup of 2.32 for multi-port memories. The difference in computation time between multi-bank and multi-port configurations was due to the DCT/IDCT. As we will see later in Chapter 10, this difference can be removed as DCT/IDCT can be computed without overhead with a bit more flexible addressing. The vector address in [12] is limited to a base address and elements following with a fixed stride between them. Moreover, in [12] they are mostly concerned with the effect of unaligned access. The VIRAM architecture [30, 31, 32] uses a memory system with similar capabilities.

CUDA devices also use multi-bank memories. Within a warp [62] (i.e. CUDA threads that are executed concurrently) the local shared memory uses multiple memory banks. The suggested approach is to use padding [63] (that we will consider in Section 6.4); however, it is not automatic. There are formalizations for automatic padding for some problems, e.g. 3D FFT [64], but the ones found in literature are problem specific.

6.3 Permutation

Using permutation for accessing data efficiently has been worked on for a long time. One early example is [65], in which different access patterns

that can be defined as certain functions are analyzed. We will formulate the problem very similarly. However, the set of addressing functions in [65] is quite limited, and as we are interested in arbitrary methods we will use a completely different approach [66].

As a simple example, we can consider a four bank memory with sixteen values as in Figure 6.1, where the numbers represent addresses. Assuming that we would want access the rows in this four by four ma-

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 6.1: Standard address layout.

trix, and also the columns, the column-wise access would cost four cycles as the column values reside in the same memory bank. However, by changing the physical location of the addresses, as in Figure 6.2, both row- and column-wise access are one-cycle operations.

The problem of finding a suitable permutation is split into two parts:

1. Find a bank assignment function $S, S : \mathbb{R} \rightarrow \mathbb{B}$, that map each address in the set of addresses $\mathbb{R}$ into the set of memory banks $\mathbb{B}$.
2. Find an address assignment function that $a, a : \mathbb{R} \rightarrow \mathbb{A}$, that maps

0	1	2	3
7	4	5	6
10	11	8	9
13	14	15	12

Figure 6.2: Permuted addresses.

each address to a unique address within its assigned memory bank memory bank.

Given these functions, a function P , as in Equation (6.3), is used to map the logical address to a physical address.

$$\begin{aligned}
 P : \mathbb{R} &\rightarrow \mathbb{A} \times \mathbb{B} \\
 r &\mapsto (a(r), S(r))
 \end{aligned}
 \tag{6.3}$$

For P to be an acceptable function that maps logical addresses to physical addresses it is required that the physical addresses are unique for each logical address, i.e. P must be a bijective function.

Definition 1. P is admissible w.r.t. $\mathbb{R}$ if Equation 6.4 holds.

$$\forall r, r' \in \mathbb{R} : r \neq r' \iff (a(r), S(r)) \neq (a(r'), S(r')). \tag{6.4}$$

I.e., no two values are allocated to the same physical address.

For conflict-free data access, we only need to concern ourselves with the function S . If it places addresses used in a parallel access in different memory banks the access will of course be conflict-free.

A parallel data access is specified as an *access format* F .

Definition 2. An access format F , with width M , is defined by a set of n -dimensional (n is the number of dimensions of the array) vectors. The access format F of width M is defined by a set of n -dimensional vectors $\vec{e}^m = (e_1^m, \dots, e_n^m)$.

$$F = \{\vec{e}^1, \vec{e}^2, \dots, \vec{e}^M\} \quad (6.5)$$

The vectors specifies the offsets from a base address r that is specific for a particular access. I.e.,

$$F(r) = \{r + \vec{e}^1, r + \vec{e}^2, \dots, r + \vec{e}^M\} \quad (6.6)$$

Definition 3. A bank assignment function $S(r)$ is conflict-free w.r.t. an access $F(r)$ if Equation (6.7) holds.

$$\forall r', r'' \in F(r) : r' \neq r'' \iff S(r') \neq S(r'') \quad (6.7)$$

I.e., no two addresses in an access should be mapped to the same memory bank.

Definition 4. $S(r)$ is conflict-free w.r.t. a set of access formats, $\mathbb{F}(r) = \{F_1(r), \dots, F_M(r)\}$, if $\forall F_m(r) \in \mathbb{F}(r)$, $S(r)$ is conflict-free w.r.t. $F_m(r)$.

As it can be very time consuming to find the optimal permutation it is useful to be able to reuse earlier solutions. It is not necessary for the problem instances to be exactly equal in order to reuse solutions. The ordering relation $\leq$ is defined in Definition (5).

Definition 5 ($\leq$). The ordering relation $\leq$ is defined for the set of all access format sets $\mathbb{A}\mathbb{F}$ as

$$\forall \mathbb{F}_i, \mathbb{F}_j \in \mathbb{A}\mathbb{F} : \mathbb{F}_i \leq \mathbb{F}_j \iff \forall F_i \in \mathbb{F}_i (\exists F_j \in \mathbb{F}_j (F_i \subseteq F_j)). \quad (6.8)$$

I.e., if a set of accesses, $\mathbb{F}_i$, only contain accesses that are subsets of other accesses, $F_i \subseteq F_j$, in another set of accesses, $\mathbb{F}_j$, it is considered smaller. Theorem 1 can then be used to decide whether or not preexisting solutions are satisfactory.

Theorem 1. If $\mathbb{F}_i \leq \mathbb{F}_j$ and $S(r)$ is conflict-free w.r.t. $\mathbb{F}_j$, then $S(r)$ is conflict-free w.r.t. $\mathbb{F}_i$.

Proof. By definition, $S(r)$ is conflict-free w.r.t. F_j in Equation 6.8. As $F_i \subseteq F_j$, $S(r)$ will not map any elements in F_i to the same memory bank, and $S(r)$ is conflict-free w.r.t. F_i as well. Again, by definition, this is true for all $F_i \in \mathbb{F}_i$, and $S(r)$ is consequently conflict-free w.r.t. $\mathbb{F}_i$. $\square$

Recall that for the LVMs we can remap logical addresses to physical addresses using functions of the kind as in Equation (6.9).

$$\begin{aligned} P : \mathbb{R} &\rightarrow \mathbb{R} \\ r &\mapsto r + p \end{aligned} \tag{6.9}$$

Our problem now is to find the permutation that minimizes the number of memory bank conflicts. I.e., given a set of desired accesses $\mathbb{X}$, we want a new set $\mathbb{Y}$ that is the same as $\mathbb{X}$ except with all the addresses remapped in a way that has as few conflicts as possible, as it is in general not possible to guarantee a conflict-free solution [65]. Therefore, we have that $|\mathbb{Y}| \geq |\mathbb{X}|$ as sometimes the actual accesses must be split into two or more accesses.

6.3.1 ILP model

We use an ILP model to find a suitable $S(r)$ function, and will defer the creation of the function $a(r)$ for later. Each address must reside in exactly one memory bank. Let $b_{r,i}$ be a decision variable (i.e. is an integer with either 0 or 1 as its value) that determines if the address r is allocated to memory bank i . Then we can constrain the model to allocate each address into exactly one memory bank using Equation (6.10).

$$\forall r \in \mathbb{R} : \sum_{i=0}^{N-1} b_{r,i} = 1. \tag{6.10}$$

The cost function is defined as the cardinality of $\mathbb{Y}$. This does not take into consideration other scheduling constraints that would potentially cause the processor to stall anyway. A data access can be completed in

the number of cycles as the maximum number of addresses that share the same memory bank. Consequently, the access costs for an access $x \in \mathbb{X}$ is

$$\forall i \in \mathbb{B} : ac_x \geq \sum_{r \in x} b_{ri}. \quad (6.11)$$

The final cost function can then be defined as

$$\min \sum_{x \in \mathbb{X}} ac_x. \quad (6.12)$$

Solving this problem gives us an easy way to generate $S(r)$ as in Equation (6.13).

$$S(r) = \sum_{i \in \mathbb{B}} b_{ri} \cdot i \quad (6.13)$$

6.3.2 Pre-processing

For most computational kernels the addressing structure is very repetitive. As it is time consuming to solve large ILP models, it is worth the effort to reduce the problem size and only solve identical problems once. This also solves the problem of not ending up with an unreasonable large amount of permutation vectors that need to be stored in memory, by reusing the same permutation vectors for many data blocks. As we want the solution to be automatic and not require any annotations by a programmer, we have developed a software tool that partitions the problem. For simplicity, all accesses are fed into a scanner that finds the substructures.

The first step is to untangle unrelated accessing problems. If we view the address nodes as an undirected graph, where nodes are connected if there exists a data access where both addresses are present, we can view the resulting connected components as different problems. If we look at the set of accesses, $\mathbb{CA}$, that belong to one connected component, we can derive a set of access formats, $\mathbb{CF}$ as in Equation (6.14).

$$\mathbb{CF} = \mathbb{CA} - \min(\mathbb{CA}) \quad (6.14)$$

Next, we sort the problems topologically using the relation defined in Definition (5). We solve the largest problems first. We know that if

$$\exists \mathbb{F}_j : \mathbb{F}_i \leq \mathbb{F}_j \wedge cf?(\mathbb{F}_j), \quad (6.15)$$

where $cf?$ is a function that returns true if a format is conflict-free, we can reuse the solution from $\mathbb{F}_j$, as it will be conflict-free for $\mathbb{F}_i$ as well according to Theorem (1). Should Equation (6.16) hold we can use $\mathbb{F}_j$'s solution as well as they are equivalent.

$$\mathbb{F}_i \leq \mathbb{F}_j \wedge \mathbb{F}_j \leq \mathbb{F}_i \quad (6.16)$$

Minimizing the Problem by Dimensional Range

One could consider to try to perform some sort of loop analysis to try to figure out which access pattern(s) that will be repeated. However, for the problem sizes of the intended application domains there is no issue generating the complete graph.

6.3.3 Post-processing

After solving the ILP model we have found $S(r)$ (Equation (6.13)). What is left is to find the function $a(r)$. The simple approach is increasing the value for the addresses mapped to each memory bank based on the original address value. I.e.,

$$r, r' \in \mathbb{R} : r < r' \wedge S(r) = S(r') \implies a(r) < a(r') \quad (6.17)$$

while minimizing the maximum values of $a(r)$ (no wasteful gaps). As ePUMA have no concept of addressing tuples of the form $\mathbb{A} \times \mathbb{B}$ the addresses are mapped into a linear space by

$$P'(r) = a(r) \cdot N + S(r). \quad (6.18)$$

This can either be done for each component separately or for all addresses. If it is done on a component basis, they are simply laid out in increasing order based on the lowest original address. We do this by adding an offset a_c to each component. Note that this may change the memory bank from what is given by $S(r)$ for the given component if

$$a_c \not\equiv 0 \pmod{N}. \quad (6.19)$$

However, this will simply rotate the memory banks in the solution and the access cost will not be changed. Also, this will not affect the number of access patterns that needs to be stored in memory.

Discussion

Even though this method can be time consuming at times, as we will see in Chapter 10, the number of actual patterns encountered in the wild is not that large. Thus, maintaining a database with solved patterns should in most cases result in more or less instantaneous solutions given that they, or perhaps more importantly another more general pattern, have been encountered before.

6.3.4 Applicability and Other Methods

The solution presented in this section works on arbitrary static access patterns. If we add some constraints, it is possible to use simpler methods. In [57] a proof that we can always find a conflict-free permutation is given, under the following assumptions:

1. The addresses of a buffer is denoted $\mathbb{A}$.
2. We can sort the read and write accesses into two sets, $\mathbb{R}$ and $\mathbb{W}$.
3. Each of the sets $\mathbb{R}$ and $\mathbb{W}$ is a set of sets. Let $\mathbb{S}$ denote either $\mathbb{R}$ or $\mathbb{W}$.
4. All of addresses in $\mathbb{A}$ is present in one and only one subset, i.e. $\forall a \in \mathbb{A} (\exists! S \in \mathbb{S} (a \in S))$.
5. The sets are of the same size that is also the size of the memory width. I.e. $\forall S \in \sim : |S| = N$.

As long as we have buffers where each value is written once and read once, we can of course reformulate it to fulfill the requirements above by using some dummy elements, and use the method in [57]. However, if we work under the assumption that each element in a buffer is written once and read once, we know that there exists a permutation that is

conflict-free and we can just as well use graph coloring to find a suitable permutation. I.e. associate each memory bank with a specific color. This is a well-studied problem and many efficient methods exist, for example the one found in [67].

6.4 Padding

One problem with full permutation is that the permutation vectors need to be stored. For some problems there is no other option, e.g. for some zig-zag patterns in some media applications, but for many algorithms with regular access patterns it is possible to trade this memory for the amount of memory the buffers themselves require [68]. One common way to solve the problem when the accesses have the format $F = f^n$, which is an abbreviation for

$$f^n = (0, f, 2 \cdot f, \dots, (n-1) \cdot f), \quad (6.20)$$

is to add dummy elements as in [68].

Assume that we have an access format N^4 . An access $F(r)$ will have an access cost of 4 as the addresses are mapped to memory bank i if

$$a \equiv i \pmod{N}. \quad (6.21)$$

Now, for one access each address can be written on the form

$$r + j \cdot N. \quad (6.22)$$

That is, each address is mapped to memory bank

$$r + j \cdot N \equiv r \pmod{N}, \quad (6.23)$$

which will be the same for all addresses. Figure 6.3 shows an example of this for $N = 8$. $F(0)$ is mapped to four addresses that reside in the same memory bank.

Now, if we would insert a dummy element after N elements, an address a would be remapped to

$$a' = \lfloor a \cdot (1 + 1 \div N) \rfloor = a + \lfloor a \div N \rfloor. \quad (6.24)$$

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
...	...	...	...	...	...	...	...

Figure 6.3: Standard address layout.

Consequently, for each address we get

$$(r + j \cdot N) + \lfloor ((r + j \cdot N) \div N) \rfloor \equiv \quad (6.25)$$

$$r + \lfloor (r \div N + j) \rfloor \equiv \quad (6.26)$$

$$r + j + \lfloor r \div N \rfloor \pmod{N}. \quad (6.27)$$

As the values for j will be unique and consecutive we will get conflict-free accesses as in $F(0)$ in Figure 6.4.

6.4.1 Problem Formulation

Let us consider a sequence of addresses that we wish to access. Assume that the accesses $\mathbf{a}_k$ for a sequence $k \in K$, where K is a set of sequences, can be written as

$$\mathbf{a}_k = \{b_k + i \cdot s + \vec{d}_k \mid 0 \leq i < I\}. \quad (6.28)$$

b_k is the start address of the sequence, i is the i :th access, s is the stride, and $\vec{d}_k$ is an access format. Assuming that we have a set of sequences accessing a buffer, how can we use padding in order to best resolve memory bank conflicts?

0	1	2	3	4	5	6	7
x	8	9	10	11	12	13	14
15	x	16	17	18	19	20	21
22	23	x	24	25	26	27	28
29	30	31	x	...	...	...	...

Figure 6.4: Address layout with padding.

6.4.2 Common Patterns

It is very common to access addresses that are equidistant from each other, i.e. straight-line access. These accesses can be expressed as

$$\mathbf{a}_k = b_k + i \cdot s + \vec{d}_k, \quad (6.29)$$

$$\vec{d}_k = d^w, \quad (6.30)$$

Another common access pattern is when we have multiple straight-line accesses. If we have n straight-line accesses of length l , we can view this as a matrix

$$\begin{pmatrix} d_1^l \\ d_2 + d_1^l \\ \vdots \\ (n-1) \cdot d_2 + d_1^l \end{pmatrix}, \quad (6.31)$$

i.e. vectors d_1^l with a distance d_2 between each other. This can be flattened to an access format to obtain $\vec{d}$. E.g. $n = 2, l = 4, d_1 = 1$, and $d_2 = 8$ would yield

$$\vec{d} = \{0, 1, 2, 3, 8, 9, 10, 11\}. \quad (6.32)$$

Luckily, as we will see later in this section, for these patterns it is always possible to find padding points that yields conflict-free data access,

at least while they're the only sequences, and we will use these padding points when searching for solutions that minimizes access costs for the set of sequences.

6.4.3 Generators for $\mathbb{Z}_n$

Finding padding points that resolves memory conflicts for straight-line access is an equivalent problem to finding a *generator* for the group $\mathbb{Z}_n$. $\mathbb{Z}_n$ is the group of natural numbers under addition modulo n . That is, it contains the elements $\{0, \dots, n-1\}$. If we have a stride s , the set generated by repeatedly adding this number is denoted $\langle s \rangle$. If $n = 8$ and $s = 2$ then we will get the sequence $2, 4, 6, 0, 2, \dots$, and we get $\langle s \rangle = \{0, 2, 4, 6\}$. This is only half of the values in $\mathbb{Z}_8$. If we would have $s = 3$ we would get $\langle s \rangle = \{3, 6, 1, 4, 7, 2, 5, 0\} = \{0, 1, 2, 3, 4, 5, 6, 7\}$. That is, we have generated all elements and 3 is then called a *generator*. When we have a constant stride, we would like the cardinality of set of values generated to be at least the size of the access width. For groups, the order $o(s)$ of a set generated is easily calculated as

$$o(s) = \frac{n}{\gcd(s, n)}, \quad (6.33)$$

where $\gcd$ is the *greatest common divisor*. s is a generator if $\gcd(s, n) = 1$, i.e. they are relatively prime.

6.4.4 Padding for Straight-Line Access

If we consider the case of straight-line access and assume that the number of memory banks is $N = 2^x$, we have that for any s

$$N = 2^x \implies \gcd(s, N) = 2^y. \quad (6.34)$$

We also know that for all elements e in the generated set

$$\forall e \in \langle s \rangle : 2^y \mid e, \quad (6.35)$$

because since s contains 2^y as a factor, any sum of them will obviously be divisible by 2^y . We know that the order of the generated set will be

$$o(s) = 2^x \div 2^y = 2^{x-y}, \quad (6.36)$$

and consequently the values generated (in some order) will be

$$\langle s \rangle = \{i \cdot 2^y \mid i \in \{0, \dots, o(s) - 1\}\}. \quad (6.37)$$

If $o(s) < N$, we know that we can take at least $o(s)$ elements without conflict. It will then be sufficient to add one dummy element every $o(s) \cdot s$ elements in order to resolve the memory bank conflicts. If we don't have conflict-free access then we have that

$$s = 2^y + s', \quad (6.38)$$

where s' is odd. We know from Equation (6.37), that

$$\forall e \in \langle s \rangle : e \equiv 0 \pmod{2^y}. \quad (6.39)$$

Including an offset r gives us

$$\forall e \in \langle s \rangle : e + r \equiv r \pmod{2^y} \quad (6.40)$$

For two different offsets, r_1 and r_2 , there will not be any overlapping values unless $r_1 \equiv r_2 \pmod{2^y}$. Since we have 2^y possible offsets that do not overlap each other, we can potentially fetch

$$2^y \cdot o(s) = 2^y \cdot \frac{N}{2^y} = N \quad (6.41)$$

values, i.e. a full vector.

Now we only have to generate the offsets. If we insert a padding of size p every $o(s) \cdot s$ elements, we can choose p so that $o(s) \cdot s + p$ generates $\mathbb{Z}_{2^y}$. As any odd number will do, $p = 1$ can be chosen. We can note that a buffer padded in this way will consume

$$\frac{s \cdot o(s) + p}{d \cdot o(s)} = 1 + \frac{p}{s \cdot o(s)} \quad (6.42)$$

additional memory, with a memory overhead of

$$\frac{p}{s \cdot o(s)} = \frac{p}{s' \cdot 2^y \cdot o(s)} = \frac{p}{s' \cdot N} \leq \frac{p}{N}. \quad (6.43)$$

6.4.5 Padding for Multiple Straight-Line Accesses

Let l be the length of the vector accesses and n the number of vectors. As we have N memory banks, we know that

$$l \cdot n \leq N. \quad (6.44)$$

For simplicity we will assume that

$$l \cdot n = N. \quad (6.45)$$

The distance between each element in a vector will be denoted as s_e and the distance between each vector as s_v .

There are two cases to consider here.

$l \leq o(s_e)$ If $l \leq o(s_e)$ then that means that we can fetch $x = o(s_e) \div l$ vectors before a bank conflict. I.e. after x vectors we should add a dummy element. The distance between vectors is s_v which gives us a padding p every $x \cdot s_v$ elements.

$l > o(s_e)$ In this case we cannot fetch even a single vector. This requires a padding p_1 every $s_e \cdot o(s_e)$ elements, such that $o(s_e \cdot o(s_e) + p_1) \geq l \div o(s_e)$. After a padding is added that ensures that one vector can be accessed, we add a second padding p_2 every s_v to ensure that each vector does not overlap each other.

6.4.6 Padding Set Evaluation

While we know how to identify certain interesting padding points for (multiple) straight-line access, we are likely to have at least two different sequences, one for reading and one for writing. Based on the suitable padding points for individual sequences, we can generate different sets of paddings. We will discuss the generation of such padding sets later in Section 6.4.7, and start with how such sets will look like and their effect on the access costs and memory consumption for a set of sequences.

A padding set is defined as a set

$$\mathbb{P} = \{(p_1, s_1), \dots, (p_n, s_n)\}, \quad (6.46)$$

where each pair p_j and s_j means that a padding of size s_j will be inserted every p_j elements. In this section we will discuss how to evaluate the effects of such a set.

One problem of padding is that the vector $\vec{d}_k$ in Equation (6.28) is not necessarily independent of i anymore, since a padding point may have been added within minimum and maximum address of access i . This is not a problem, as we can rewrite the sequences as

$$\mathbf{a}_k = \{b_k + i \cdot s + \vec{d}_i \mid 0 \leq i < I\}, \quad (6.47)$$

and use the permutation and addressing features of the MPEs to handle this case. The question is then, how many permutation vectors do we need?

Looking at just one padding j , the vectors would start to repeat after

$$c = \frac{lcm(s, p_j)}{s} \quad (6.48)$$

iterations. Computing the least common multiple between the padding distance p_j and the stride s we know how many elements there are between each time they line up, and dividing that by the stride s gives us the number of iterations before repetition occurs. We will obviously require the same amount of permutation vectors. However, in order to be able to reuse the permutation vectors we will need to ensure that the address register that we use for traversing the data structure is updated with the padding added. If we calculate the least common multiple for p_j and s , and instead divide with p_j rather than s we will get the number of padding points between each synchronization point. Multiplying this with the size of the padding s_j will give us the total amount of added dummy elements. That is,

$$\text{extra elements} = \frac{lcm(s, p_i)}{p_i} \cdot s_i. \quad (6.49)$$

Adding this to the address register used every c iterations will ensure that the address register has a value that is compatible with the permutation vectors.

In some cases it is possible to shrink the number of vectors used a bit more. Specifically, if $s \mid p_j$ (i.e. $p_j \equiv 0 \pmod{s}$) it is possible to just use one vector if

$$\forall e, s \geq 0 : b + d_e \bmod p_j < s. \quad (6.50)$$

d_e denotes one value in the access vector. The reason for this is that if that is true, then

$$(b + d_e \bmod p_j) + (c - 1) \cdot s < s + (c - 1) \cdot s = c \cdot s = p_j. \quad (6.51)$$

In other words, all elements d_e will be used to reference will be between two padding boundaries, and it will not need an adjustment. In a similar way we only need one vector if $s \mid p_j$ and

$$\forall e, s < 0 : b + d_e \bmod p_j \geq s + p_j. \quad (6.52)$$

We can do the same with a full set of paddings. Let

$$\alpha = \text{lcm}(s, p_1, \dots, p_n), \quad (6.53)$$

$$\alpha = c_\alpha \cdot s. \quad (6.54)$$

The number of iterations before everything synchronizes is then c_α . If we also assume that removing one padding frequency from the set gives us

$$\beta = \text{lcm}(s, p_1, \dots, p_{n-1}), \quad (6.55)$$

$$\beta = j_\beta \cdot s, \quad (6.56)$$

and that

$$\beta \mid \alpha, \quad (6.57)$$

$$\forall i, e \in I \times E : b + d_e \bmod p_i < s. \quad (6.58)$$

In this case d_e will move over a padding boundary, and we only need to update the address register every c_α iteration with respect to (s_n, p_n) . We may of course repeat this until this condition no longer holds for any padding frequency left. When this process is completed we will find that we have c_γ necessary vectors and that we have a set of accesses

$\mathbb{A} = \{b_k + i \cdot s + \vec{a}_i \mid i \in 0, \dots, j_\gamma - 1\}$, that can be repeated after c_γ iterations to form the complete access sequence. As an address a is moved to address a_p ,

$$a_p = a + \sum_{i=1}^n \left\lfloor \frac{a}{p_i} \right\rfloor \cdot s_i, \quad (6.59)$$

it is easy to calculate the necessary permutation vectors.

As we will generate the permutation vectors for all sequences and for all buffers in a kernel, it is not unusual that many are identical or that some list of permutation vectors are sublists of larger lists. By removing lists that are sublists of other lists the number of vectors are reduced further. Another effect of padding data buffers is that the actual data memory cost is increased. We can quite easily calculate the cost of one padding p_j with

$$v = \begin{cases} 1 & \text{if } p_i \mid \text{size} \\ 0 & \text{if } p_i \nmid \text{size} \end{cases} \quad (6.60)$$

$$\text{Memory cost } p_{mc} = \left(\left\lfloor \frac{\text{buffer size}}{p_j} \right\rfloor - v \right) \cdot s_j. \quad (6.61)$$

This yields a total memory cost of

$$\sum_{i=1}^n p_{mc}^{(i)}. \quad (6.62)$$

6.4.7 Generating Valid Padding Sets

Naturally, we want to find the best padding set that allows for the fastest execution time without using more resources than are available in the system. Trying all of the possible padding distances and sizes is infeasible, so it is necessary to try to trim the search space in some manner. Considering the padding points for conflict-free access in Section 6.4.5, we can see that the interesting distances are related to the stride within an access and the outer stride. The implementation used for ePUMA uses those strides and multiples of those strides that are smaller than the buffer. We refer to this set of strides as *strides*. We can also note that the padding sizes should be smaller than N , as it makes no difference w.r.t.

whether it becomes conflict-free or not as the memory banks are calculated using $\text{mod } N$. We can also note that as some padding distances may be multiples of others, it may be possible to have negative sizes. Given this, the set of possible strides is

$$\Omega = \{(p_i, s_i) \mid p_i \in \text{strides}, s_i < N\}. \quad (6.63)$$

A valid padding set is drawn from the powerset of Ω , with the constraint that the sum of padding sizes at specific points are not negative, as in Equation (6.64).

$$\mathbb{P}_{\text{valid}} \in \{\mathbb{P} \in \wp(\Omega) \mid \sum_{j \text{ s.t. } p_j \mid p_i} s_j \geq 0\} \quad (6.64)$$

There is seldom any need to look at solution with too many padding points, as the good solutions tend to use few. Setting up a condition of

$$|\mathbb{P}| \leq t, \quad (6.65)$$

where t is parameter to control the size of the final padding sets, tends to find the best valid solutions in Ω without having a value larger than 3 or 4. For these sizes generating a Pareto set is quite quick. For our problems, choosing the fastest solution has been acceptable from a memory resource point of view. However, as this is calculated per buffer the selection process could become slightly more difficult. However, as there aren't usually that many solutions in the Pareto set, one could just enumerate all possibilities or formulate an ILP program if enumeration would take too long.

6.5 Rotation

Rather than using padding we could possibly rotate different segments of a buffer instead to get the same effect. Considering the case represented by Figure 6.3 again, we could use an address layout as shown in Figure 6.5 instead. This way we do not expend data memory. I.e., we can consider a rotation set

$$\mathbb{R} = \{(r_1, s_1), \dots, (r_n, s_n)\}, \quad (6.66)$$

0	1	2	3	4	5	6	7
15	8	9	10	11	12	13	14
22	23	16	17	18	19	20	21
29	30	31	24	25	26	27	28
...	...	...	...	...	...	...	...

Figure 6.5: Address layout with rotation.

rather than a padding set

$$\mathbb{P} = \{(p_1, s_1), \dots, (p_n, s_n)\}. \quad (6.67)$$

Assume that if

$$r_i < r_j, \text{ then} \quad (6.68)$$

$$r_i \mid r_j. \quad (6.69)$$

The effect of one rotation is defined in Equation (6.70).

$$\text{rotate}(a, r, s) = \left\lfloor \frac{a}{r} \right\rfloor \cdot r + \left(a + \left\lfloor \frac{a}{r} \right\rfloor \cdot s \right) \bmod r \quad (6.70)$$

The final address can then be calculated using the algorithm in Listing 2. The problem with this approach is that we will need more permutation vectors. Each sequence starting address will result in a unique table of permutation vectors as a different amount of elements have been shifted around the “edge”.

Algorithm 2 Calculate address

```

1: function CALCULATE_ADDRESS( $a, \mathbb{R}$ )
2:    $r, s \leftarrow \min \mathbb{R}$ 
3:    $\mathbb{R} \leftarrow \mathbb{R} - (r, s)$ 
4:   while  $\mathbb{R} \neq \emptyset$  do
5:      $a \leftarrow rotate(a, r, s)$ 
6:   return  $a$ 

```

6.6 Using Non-power of 2 Number of Memory Banks

When using strided access, we can note that the problem of memory bank conflicts occurs when the factors in N overlaps with the stride. I.e., if we use a number of memory banks that is relatively prime with the strides we would not have memory bank conflicts. By determining the most commons strides for a particular application, a suitable memory configuration could be selected. There is however the cost of calculating the memory bank. With a normal layout we would need to know the address in a bank

$$\text{address in bank} = \frac{\text{address}}{N}, \quad (6.71)$$

and the memory bank

$$\text{bank} = \text{address mod } N. \quad (6.72)$$

For an N that is a power of two this is very simple, as the address is represented in base 2, but is more expensive for other powers, since hardware for division and modulo would be required. It would however not be as expensive as general division, as the number of banks N would be fixed.

It should also be noted that it is only the memory bank calculation that is necessary, i.e.

$$\text{bank} = \text{address mod } N. \quad (6.73)$$

If the size of each memory bank is M , then if N and M are relatively

prime all addresses will be generated if we just use

$$\text{address in bank} = \frac{\text{address}}{M}. \quad (6.74)$$

M would quite naturally be chosen to be a power of two, making it very easy to compute. This is covered in more detail in [65].

Another issue with a non-power of 2 multi-bank memory is how to support data types of different size. If we would choose something suitable for 32 bit data we would like the number of banks to be even, as we need to fit two words for each value. This would lead to issues when using 16 bit values with strides of a power of two, and would require methods like in Section 6.4 anyway.

6.7 Using Dynamic Bank Assignment Bits

Normally the least significant bits of a memory address are used for deciding memory bank. However, it doesn't really matter which bits we use. For example, if we have a stride of 2^n and eight memory banks, then using bits $n+2, n+1, n$ would generate conflict-free bank assignments. This would even work on a per buffer level as long as the buffers would be aligned and, quite naturally, that the buffer size is larger or equal to 2^{n+3} . This will not really solve the problem either, as it is quite common that we have accesses with strides 2^n and 2^{n+1} , e.g. for write/read, on a buffer. The FFT is one example of an algorithm with this property. Only one would have conflict-free access, while the other would have a cost of two for each access.

6.8 Permutation of Dimensions

It should be noted that many problems can be solved by simply swapping the layout of buffers. E.g., if we would have a buffer with dimensions

$$R_1 \times R_2, \quad (6.75)$$

and R_2 is a power of 2, then column-wise access would not be possible. Let's say that R_1 is odd. In this case a buffer of the shape

$$R_2 \times R_1 \tag{6.76}$$

would enable both column- and row-wise access. As this version is isomorphic to the first version, the only change needed is the calculation of the strides for assembly programs. For programs in higher level languages this can be fully automatic. This can be used for C-like structs as well, as they can be viewed as arrays. For example, an array of length R of a struct of W words,

$$R \times W \tag{6.77}$$

can just as well be represented as W arrays of length R .

$$W \times R. \tag{6.78}$$

6.9 Summary

In this chapter we have (mainly) considered two methods for solving the parallel data access problem for static access patterns. First, we considered the use of full permutation, that is the most general form. While it will find the best data address layout possible, the cost of the number of permutation vectors may be prohibitive, and finding the solution can at times be too time consuming.

A second approach based on padding has also been considered, that uses extra data memory to reduce the data access costs. While this approach is more limited, for most problems the limitations are not an issue.

Chapter 7

Code Generation for MPEs

In this chapter we will consider the problem of utilizing the addressing and vector operation features of the MPEs, and memory placement. Given that the problem of transforming data is solved and we can perform overhead-free vectorized data access on buffers in local memory, we still have a few issues to deal with. There are primarily two issues:

Vector Instruction Utilization The first problem is understanding from source code that some computation may be vectorized. This information can be extracted either by analysis of the source code or using a language with parallel constructs built in.

Memory management The second problem is memory management. The MPEs can work on multiple memories simultaneously to support memory parallelism. While a buffer may be configured in such a way that data may be accessed in parallel, if two buffers are located in the same single port memory there will only be at most one vector fetched every clock cycle. Consequently, we want to distribute the buffers over the memories in such a way that we rarely end up in this situation. Also, if it is unavoidable, we would like the buffers that cause the least overhead to be the ones that are placed in the same memory.

Our solution uses a Domain Specific Language (DSL) called FT, where vector operations are primitives available in the language. This way we

get both information of which operations that may be performed in parallel, and information on how the different buffers in memory are addressed.

7.1 FT

To explore the possibilities of compiling code for the MPEs, a Domain Specific Language (DSL) called FT¹ has been developed. A DSL is simply put a language similar to more general languages, but with constructs that suit the application domain, e.g. SQL for working with databases. The purpose of FT is not to design a new language, it is simply used as a vehicle for exploring compilation on ePUMA, and figuring out what kind of abstractions are necessary for efficient compilation. While there are many compilation methods available [69], none of them are directly suitable for utilizing the specific features of ePUMA. The need for special handling for ePUMA is not unique, it is a general trend for different high performance computing systems [70]. For example, even though a lot of work has been put into vectorization (e.g. [71, 72, 69, 73]), very few loops from real applications are vectorized [74]. Therefore, writing code in a language where vector operations are first class primitives is a better idea [70].² This is not a new idea, e.g. ELP [75] is more than two decades old, and even before that compiling e.g. APL for vector processors was considered [76]. However, rather than writing all code in a new language, it is also quite common to generate code from some other specification. For example, the Spiral project uses a language called SPL (Signal Processing Language), a symbolic language for formulas that is translated into C with macros that uses intrinsics for utilizing vector instructions [77].

We will not concern ourselves with trying to match the most suitable instructions, and the more specific instructions for the platform are avail-

¹FT is a smaller abbreviation of Formula Translation than Fortran is. FT shares many design goals with Fortran but with a much smaller application scope.

²Given that we are concerned with efficient code generation, rather than dealing with legacy code.

able as intrinsics. While instruction selection is an important problem it is not ePUMA specific. However, one way would be to use e.g. constraint programming [78].

7.1.1 FT Features

To be a bit more specific, what does FT offer that e.g. C does not?

Vector operations Similarly to e.g. Matlab, FT can operate on vectors, as in Listing 7.1. If we have two vectors of the same shape we can do element-wise multiplication by just multiplying them, and if one is e.g. a scalar that value scales the other vector.

Reduction operations In many functional programming languages there is a standard library function called *reduce*, that takes a binary operator, an initial value and a collection, and then successively accumulates a result as it traverses the collection, e.g.

$$\text{reduce } (+) \ 0 \ [1, 2, 3] = (((0 + 1) + 2) + 3). \quad (7.1)$$

By matching on certain patterns, as e.g. in Listing 7.2, we can very easily deduce that we can use the MAC instructions and available registers for fast computation. I.e. an element-wise multiplication of two vectors is performed and then reduced with addition, which the MPEs can execute very efficiently.

reduce is just one example of a higher-order function, and FT was not designed to be limited to a specific set of higher-order functions. However, *reduce* is a typical example where we can provide very efficient implementations.

Explicit addressing We can extract the addressing patterns easily without any complicated analysis. As seen in Listing 7.3, we can apply different addressing patterns directly to the operands, e.g. choosing every second item, or index the values with some other array. As these patterns can execute efficiently on the MPEs, they along with other methods of specifying the vectors operated on are included on a language level.

Listing 7.1: Vector multiplication

```
1 let xs : int[n]
2 let ys : int[n]
3 let zs : int[n]
4 let y  : int
5
6 zs = xs * ys // Element-wise multiplication
7 zs = xs * y  // Scale the values in xs with y
```

Listing 7.2: Vector MAC

```
1 let xs : int[n]
2 let ys : int[n]
3 let z  : int
4
5 z = reduce (+) 0 (xs * ys)
6 z = sum (xs * ys) // Equivalent statement
```

Intrinsics Special functions can be used without overhead. E.g., if we consider Listing 7.4, a layer of an FFT can be implemented as simply as issuing a function, without any of the overhead.

Data types As the numerical types for DSP often is different from mainstream languages, e.g. fix-point representations are often used, it is possible to have support this and let the compiler track the different representations.

So what are the benefits from using a DSL?

Abstraction As many algorithms are prototyped in languages such as Matlab, there is no need to rewrite it as an over-specified C program, and instead use primitives from languages with higher levels of abstraction.

Efficiency By primarily including language constructs that can be translated to fast machine code, we ensure that we most often achieve high efficiency.

Listing 7.3: Explicit addressing

```

1 let xs : int[n]
2 let ys : int[n]
3 let reverse_permutation : int[n]
4 reverse = [n-1,n-2,...,1,0]
5
6 let even_idx = xs[0 .. 2 .. n]
7 let odd_idx  = xs[1 .. 2 .. n]
8 let reversed = xs[reverse_permutation]

```

Listing 7.4: Intrinsic

```

1 let ys = butterfly-r2 xs coeffs

```

7.2 Memory Management

While we want to distribute arrays so that we reduce the number of memory conflicts as much as possible, we must also ensure that the number of arrays that are alive during any point of execution do not overlap and also fit into the memories. This is not very difficult to formulate as an ILP problem. Given that we have analyzed the liveness of the arrays, i.e. we know for each instruction which arrays it operates on and which other buffers that are alive, we can easily construct a model for this problem.

7.2.1 Port Mapping

The first issue to deal with is which memory an array should reside in. First, we associate each variable v (in V , the set of all variables in the program) with an array of decision variables, where the length of the array is determined by the number of memories (M). I.e.,

$$\forall v, m \in V \times M : v_m \in \{0, 1\}. \quad (7.2)$$

, where $v_m = 1$ means that variable v is mapped to memory m . Each variable must map to one specific memory.

$$\forall v \in V : \sum_{m \in M} v_m = 1 \quad (7.3)$$

For the set of variables with explicit memories assignments, EVM , we add

$$\forall v, m \in EVM : v_m = 1. \quad (7.4)$$

In the next step, we will add the costs for different accesses and minimize the total access cost. This step assumes that we will be able to generate conflict-free vector accesses later, and we only consider the cost due to memory assignment conflicts.

Consider the set N that consists of all the nodes in the intermediate representation, i.e. the representation of the program used by FT. For each such node and memory, we have that the cost for using that memory is greater than the sum of the cost of all the vector references that uses the memory.

$$\forall n, m \in N \times M : cost_{nm} \geq \sum_{vr \in vrs(n)} vr_c \cdot v_m \quad (7.5)$$

vr_c is the cost of a vector reference to vector v if no other vector is fetched from the same memory at the same time. The cost of the vector reference vr for this node, $cost_n m$, must be larger than that if the memory is used for the vector v .

The cost for a node $cost_n$ is then the maximum of the cost for each memory.

$$\forall n \in N : cost_n \geq \sum_{m \in M} cost_{nm} \quad (7.6)$$

7.2.2 Variable Address Allocation

For simplicity, we can formulate an ILP problem for the memory allocation as well. We have that

$$\forall v \in V : 0 \leq v_b < MS - v_s, \quad (7.7)$$

where v_b is the base address of variable v , v_s is its size, and MS is the memory size. Then we can successively add non-overlapping constraints as in Listing 3.

Algorithm 3 Add non-overlapping constraints.

```

1: procedure F( $V$ )
2:   while  $V_s \neq \emptyset$  do
3:      $v \leftarrow \text{any}(V)$ 
4:      $V_s \leftarrow V - v$ 
5:      $vs \leftarrow \{v' \in V \mid \exists n \in N : \text{alive}(n, v) \wedge \text{alive}(n, v') \wedge v_p = v'_p\}$ 
6:     for all  $v' \in vs$  do
7:        $\text{add\_non\_overlap}(v, v')$ 

```

7.3 Vector Instruction Utilization

The MPEs are vector processors that can operate over matrices and other 2 dimensional structures, and for large data parallel problems we naturally have good performance. However, for smaller problems the set up can dominate the computation time. As a rule of thumb, on the MPEs an operation can be considered small if

$$\frac{\text{vector length}}{\text{SIMD width}} \lesssim 10. \quad (7.8)$$

For these operations the overhead can be substantial if one does not use the addressing features in a proper way.

If we consider the execution time for one vector instruction, it is

$$\text{execution time} = o + \left\lceil \frac{n}{W} \right\rceil. \quad (7.9)$$

Here, W is the width of the data path, o is the overhead, and n is the length of the vector operation. That means that the execution time per element is

$$\text{execution time/element} = \frac{o}{n} + \frac{1}{n} \cdot \left\lceil \frac{n}{W} \right\rceil \quad (7.10)$$

If we take the limit of this, we get

$$\lim_{n \rightarrow \infty} \frac{o}{n} + \frac{1}{n} \cdot \left\lceil \frac{n}{W} \right\rceil = \frac{1}{W}. \quad (7.11)$$

For long vectors we will quite obviously get high utilization. However, if we assume a vector operation of length $n = W$ and that we have a constant overhead of $1 \div o = 1$, the utilization goes down to 50%. Even worse, a single cycle of constant overhead is often optimistic, as there is usually a delay between the instructions that set up the address and step registers and when the registers are ready to be used, in addition to the execution of the additional instructions. Even in the best case for operations on such small vectors the performance is horrible. This is a typical example of Amdahl's law [7].

7.3.1 Problem Definition

First of all, it is important to understand what kind of Control Flow Graphs (CFGs) that we can map to the MPE hardware efficiently w.r.t. iteration and addressing.

If we consider a loop such as the one in Figure 7.1, if i is an induction variable we can trivially transform this into a something that uses a *repeat* instruction instead. This transformation can be seen in Figure 7.2. By doing this, we have eliminated the need for the back jump and the check for exit condition. Now, if we have some code that uses vector instructions, as in Listing 7.5, we will get a graph as in Figure 7.3. $0 \dots s \dots I$ means that the loop will be executed with $i = 0, s, 2 \cdot s, \dots$ until $n \cdot s \geq I$. While we can use vector instructions to execute this code quite efficiently, there is still a step of updating the address register. When executing vector instructions one address register is used per vector, and the registers are automatically updated to enable the overhead free iteration. However, we must still issue the code to set up the address registers before issuing the vector instruction.

Listing 7.5: Adding multiple vectors.

```
1 for i in 0 .. s .. I do
```

Figure 7.1: A loop.

2	$X[i, :] := Y[i, :] + Z[i, :]$
---	--------------------------------

This pattern is so common that the MPEs have hardware support for this, called 2D addressing, and is the very reason that they are called Matrix Processing Elements. Essentially, it allows us to remove the overhead of two levels of iteration. We will see some examples of this in Chapter 10. While we have vector operations in one dimension, we can update the address registers arbitrarily after each vector operation, avoid the set up cost for the next instruction, and operate on for example 2D data sets. Using this feature the code may be transformed into a graph as shown in Figure 7.4.

This can now be even further reduced, as single instructions can be repeated an arbitrary number of times, and we end up with a graph as shown in Figure 7.5.

Figure 7.2: A loop using repeat.

Figure 7.3: Adding rows.

Figure 7.4: With 2D addressing.

Figure 7.5: With iteration.

7.3.2 Dataflow Analysis

In order to be able to use the 2D addressing feature of the MPE we need to be able to determine when 2D addressing is applicable. In order to do this we will use the standard dataflow analysis framework. There are two things we need to know to be able to issue instructions using 2D addressing.

1. What will an vector instruction set the address register to after it has finished executing?
2. What does any successor instructions require the address register to be set to before executing?

If we know this, it is quite trivial to actually generate the code. We simply need to apply a stride s that is equal to

$$s = \text{address required by successor} - \text{address after current instruction.} \quad (7.12)$$

7.3.3 General Idea

Assume that we have a graph that looks like in Figure 7.6, where each node contains the required address registers and register values necessary before issuing the instruction. By letting the values flow backwards, we will eventually get something like Figure 7.7. That is, each node contains the desired address for the addresses of the subsequent nodes. After this step, since each node will know what any successor nodes require an address register to be set to, and using a forward flow, we can simply propagate what the address registers are set to after each instruction. If it is possible to calculate a stride s (that is, the difference between the addresses is fixed), we add this stride to the address register by using the 2D addressing feature. This will lead to a graph like in Figure 7.8. With this information, we can note that address register arX is set to the value expected for instruction 3 by instruction 2, and we can ignore generating setup code before issuing that instruction.

Figure 7.6: Flow graph with desired addresses.

Figure 7.7: The outcome of propagating the desired addresses.

Figure 7.8: The outcome of propagating the defined values.

7.4 Transfer & Merge Functions

This section will describe the necessary transfer and merge functions for dataflow analysis. We will use the notation (r, x) for a register r and its value x . Undefined values will be denoted $\perp$ and multiple definitions $\top$. A register with multiple value definitions is in a way undefined, but must be separated from the zero definition case in order to get a monotonic set of *transfer* and *merge* functions. The functions in this section is defined on one specific register, but the total functions is of course over the entire set of address registers.

7.4.1 Address register requirements

We begin by describing the function used for propagating the required addresses, and we will start with the transfer function. If an instruction requires registers to be set, we use ΔT to denote the changes required to the state T . If a register is required to be set, it should be overwritten (Equation (7.13a)). If it is not in the set of changes, we must keep the value from some earlier successor (Equation (7.13b)). Otherwise, it is quite naturally undefined (Equation (7.13c)).

$$tf(T, \Delta T) = \begin{cases} (r, x) & \text{if } (r, x) \in \Delta T, \\ (r, y) & \text{if } (r, y) \in T, \\ (r, \perp) & \text{otherwise.} \end{cases} \quad \begin{matrix} (7.13a) \\ (7.13b) \\ (7.13c) \end{matrix}$$

When merging states, a register can have a defined request if both states have the same value for the register (Equation (7.14a)). If one state is from the body of a loop $T \in FL$ (From Loop), it has priority and generates a defined value as in Equation (7.14b), even if we would have two different values in both states. If no state is from a loop we propagate $\top$ as the value, if any state has an value is defined as $\top$ (Equation (7.14c)), or if the values are different (Equation (7.14d)). If just one state has a defined value, as in Equation (7.14e), we propagate that one. In all other

cases, the register is considered undefined (Equation (7.14f)).

$$\text{merge}(T, T') = \begin{cases} (r, x) & \text{if } (r, x) \in T \cap T', & (7.14a) \\ (r, x) & \text{if } (r, x) \in T, T \in FL, & (7.14b) \\ (r, \top) & \text{if } (r, \top) \in T \cup T', & (7.14c) \\ (r, \top) & \text{if } (r, x) \in T, (r, y) \in T', & (7.14d) \\ (r, x) & \text{if } (r, x) \in T \cup T', & (7.14e) \\ (r, \perp) & \text{otherwise.} & (7.14f) \end{cases}$$

7.4.2 Propagation of definitions

We let ΔD_i (i.e. updates of one instruction) be defined as in Equation (7.15).

$$\Delta D_i = \{(r, t) \mid r \in R(i), (r, t) \in T_i, t \notin \{\perp, \top\}\} \quad (7.15)$$

If register r

1. belongs to $R(i)$, the set of address registers used by an instruction, and
2. there is a request that r is assigned the target value t ,

then r will be set to t .

We can define the transfer function as in Equation (7.16). If an instruction will update an register, the register is set according to Equation (7.16a). If not, it is kept if it is previously defined as in Equation (7.16b), otherwise it is still undefined, as in Equation (7.16c).

$$tf(D, \Delta D) = \begin{cases} (r, x) & \text{if } (r, x) \in \Delta D, & (7.16a) \\ (r, y) & \text{if } (r, y) \in D, & (7.16b) \\ (r, \perp) & \text{otherwise.} & (7.16c) \end{cases}$$

For merging we use Equation (7.17). A register is only defined if both states have it defined as the same value (Equation (7.17a)), otherwise it is undefined (Equation (7.17b)).

$$\text{merge}(D, D') = \begin{cases} (r, x) & \text{if } (r, x) \in D \cap D', & (7.17a) \\ (r, \perp) & \text{otherwise.} & (7.17b) \end{cases}$$

7.4.3 Other optimizations

As most of the computation time is spent in loops we will insert special fix nodes before loops, as in Figure 7.9. These nodes will have special transfer functions that ensure that all loops have their required states set before entering, so that we minimize the registers that needs to be set in the loop body.

Figure 7.9: Loop with a fix node inserted.

In a loop body, address registers will in general be updated for the next iteration. Therefore it can be beneficial to extract the last iteration, so that it may set the registers correctly for the next segment of code as in Figure 7.10. This is especially useful when we have nested loops as in Figure 7.11, as it can then start the next outer iteration without initialization.

In order to get rid of the normal setup code we rely on code hoisting and dead code elimination. These are implemented in the standard ways described in e.g. [47], [45], or [46].

Figure 7.10: Last iteration moved from the loop.

Figure 7.11: Nested loops.

7.5 Summary

In this chapter we have considered code generation for the MPEs. We have covered why DSLs are suitable, the issue of allocating data to the local vector memories, and how to utilize the 2D addressing features provided by the MPEs.

Part III

**Software Development for
ePUMA**

Chapter 8

Programming Model

In this chapter we will cover the overall programming flow of ePUMA. Much of the overall process is similar to other architectures with multi-processors that have distributed memory. However, with ePUMA there are additional challenges that needs to be addressed.

8.1 General programming flow

The goal with the different software development tools for ePUMA is to support a programming environment that does not introduce any significant increase to the application development cost compared with other environments. For DSP the amount of different computationally heavy functions is not that large, and for ePUMA we propose a partitioning between kernel development, and application development. The idea is to hide the complexity of kernel implementation from the application programmers, and to provide a library with DSP functions (e.g. similar to TI's kernel library). The ePUMA APE is ARM compatible and can be programmed using mainstream languages. Programs are written as normal programs, and for performance critical parts, kernels are called with the same interface as normal functions.

8.2 Kernel level development

The ePUMA architecture enables task and data parallelism. Task parallelism is achieved by running several tasks on several cores. For example, if we have a task graph as in shown in Figure 8.1, where task $T1$ and $T2$ are two independent of tasks, then $T1$ and $T2$ can obviously be run in parallel on different cores. With the task graph in Figure 8.2 we can process the result from running $T1$ on input block n on $T2$, while $T1$ processes input block $n + 1$.¹ For the data parallel case each core simply

Figure 8.1: Task graph with two parallel tasks.

Figure 8.2: Task graph with sequential tasks that may be pipelined.

executes the same function on different input data blocks.

ePUMA programs are kernel based similarly to OpenCL [79], in contrast to the parallel programming models provided by OpenMP [80] and MPI [81] that are quite popular in current high performance computing for general-purpose processors. OpenMP’s model of computation with shared memory does not map as cleanly as the kernel based approaches

¹Of course, for both cases there are some issues if the execution times are wildly different, and the associated data transfer cost.

to the ePUMA architecture, where each MPE has its own private memory. MPI is also a poor fit as the MPEs processors are not designed to function as general purpose nodes. The APE is responsible for setting up the connections between nodes, contrary to MPI where the nodes themselves decide whether to send data or not.

When developing software for kernels, the tasks should be decomposed into as small units as possible. The tasks should have the shape as described in Chapter 5. Each task can then be implemented either in assembly or in FT.

In the ePUMA programming model, a normal C program is executed on the APE, and the heavy duty processing is offloaded to the compute clusters. The data that should be operated on should be split into blocks (blocks are described in Section 8.2.1). E.g. Figure 8.3 shows a two dimensional split with sections of 3×5 values constituting one block. This is similar to how OpenCL manages the partitioning of computation. Kernels that operate on a block should behave the same way re-

Figure 8.3: 2D blocking

gardless of order of the processing of blocks. Some optimizations may cause several kernels to run at the same time; i.e. when input into the scheduler there would be no guarantee in what order the blocks are processed. However the observed result is identical to the sequential case if

the kernels behave as pure functions. The purpose of this is that we can use a scheduling tool as in Chapter 5 without running into problems due to dependencies. This also helps with limiting the size of the problem that the scheduler needs to consider.

A kernel contains a kernel prolog (*kp*) and epilog (*ke*) that are executed on the APE. The kernel prolog is executed before any of the blocks and the epilog is executed after all blocks have finished executing. The prolog loads the program code and constant data. The epilog may e.g. be used for gathering the result of the computation if this should not be done during the computation.

8.2.1 Blocks

As previously mentioned, each kernel is divided into blocks. Each block may use up to the number of compute clusters available in the system. Communication between compute clusters are allowed within one block, while interblock communication is forbidden. This constraint is due to the fact that blocks execute until they finish and trying to communicate with some block that is not currently scheduled would result in a deadlock.

A kernel's operation on a block is viewed as a function that maps its input data to its output data. Each block has a type that specifies the properties of its input and output, e.g. size and layout. This is used for pipelining and fusing blocks as described later in Section 8.2.2 and 8.2.3. In addition to this, each block is associated with a block prolog and epilog, that are executed on the APE. They are responsible for the DMA transfers and control of blocks. The block prologs/epilogs are executed in parallel with the blocks in order to hide the block control and DMA transfer overhead. I.e. the prolog of block $n + 1$, block n and the epilog of block $n - 1$ are executed in parallel. Figure 8.4 illustrates this. Looking at a block n , its prolog is executed at the same time as the previous block and its epilog is executed during the following block, making use of the multiple memories provided by the memory subsystem. If the compute-to-data-transfer ratio is large enough, DMA transfer costs are hidden at

steady state.

The blocks in Figure 8.4 can use several compute clusters. It is possible for several blocks to execute at the same time if there is a sufficient number of CCs available. In fact, most data parallel problems should use one CC per block and consequently have n blocks in parallel on a n CC machine.

More formally, there are a few rules for the order of execution of a block and its prolog and epilog. Let p_i and e_i be the prolog and epilog for the block b_i and that $x \rightarrow y$ means that executable unit x is executed and finished before executable unit y begins. Then the following properties are guaranteed.

- $kp \rightarrow p_i$
- $p_i \rightarrow b_i$
- $b_i \rightarrow e_i$
- $e_i \rightarrow ke$

No guarantees are made by the scheduler about in which order the blocks are executed.

8.2.2 Pipelining

It is often the case that data is processed in a pipelined fashion. Some kernels create intermediary data that is only used by one of the following kernels. As some algorithms may not scale up to all the available cores, it may be sensible to run two or more kernels at the same time. Consider Figure 8.5; assume that block x from kernel f takes some source data block $SDBx$ and outputs intermediate data block $IDBx$ to memory, while block x from kernel g reads the intermediate data $IDBx$ as input data. Under the condition that $IDBx$ is not used anywhere else, and f 's blocks output size is compatible with g 's blocks input sizes, we may transform the computation as shown in Figure 8.6. This may be performed by some optimization stage or by using directives.

Figure 8.4: Kernel execution flow for blocks

8.2.3 Block composition

As blocks are typed (as described in Section 8.2.1) and provide functions such as those in Equation 8.1 and 8.2, it is quite natural to apply the function composition operator on them, and replace two kernels with one kernel whose blocks provide the mapping in Equation 8.3 instead.

$$f : \mathbb{X} \rightarrow \mathbb{Y} \quad (8.1)$$

$$g : \mathbb{Y} \rightarrow \mathbb{Z} \quad (8.2)$$

$$g \circ f : \mathbb{X} \rightarrow \mathbb{Z} \quad (8.3)$$

If we have the same situation as in Figure 8.5, and perform block composition on the kernels the result of applying block composition is shown in Figure 8.7. This is similar to the pipelining case except that the blocks have been fused into one and execute on one core instead of two, and as a result the DMA transfers between the different types of blocks have been removed. This may help with utilization since the ratio between

Figure 8.5: Execution of two kernel with intermediate data

computation and DMA transfers is increased. As with pipelining, this may be performed by an optimization stage or by using directives.

8.2.4 Kernel templates

A programming environment needs to provide rich libraries to be considered programmer friendly. Providing kernels for all possible invocations is not reasonable. To enable a prepared kernel to be used as much as possible, configurability is introduced.

A kernel template is parameterized kernel to tune the assembly or FT programs for specific invocations of algorithms. In our case, dimensions of configurations consist of at least data size, times of iteration, initial values. A configuration may also change the underlying algorithm of a kernel functions. E.g. choose different algorithms depending on input size. The result of a configuration applied a kernel template is a kernel.

Figure 8.6: Pipelined execution of two kernels

8.2.5 Data access templates

It is impossible for any library writer to anticipate all possible kernels that should be available. It is however preferable if application programmers are not required to write what is often the most difficult part for efficient execution of any algorithm; the data access code.

Many algorithms share data access patterns, and only differ in the function they apply to the data. Data access templates are identical to kernel templates, except that they also accept code as a parameter. This allows library writers to write highly optimized code for data access, leaving the application programmer with only needing to supply the function. E.g. consider the functions in Equation 8.4 and 8.5, where the only difference is that Σ uses addition and Π uses multiplication.

$$f : \mathbb{Z}^{\mathbb{N}} \rightarrow \mathbb{Z} \quad (8.4)$$

$$\{z_i | i \in 1, \dots, n\} \mapsto \sum_{i=1}^n z_i$$

Figure 8.7: Block composition of two kernels, $h = g \circ f$

$$g : \mathbb{Z}^{\mathbb{N}} \rightarrow \mathbb{Z} \quad (8.5)$$

$$\{z_i | i \in 1, \dots, n\} \mapsto \prod_{i=1}^n z_i$$

Implementing these separately would mean re-implementing the same data access pattern twice. Separating the data access from the functions into a different function called *reduce*, with the type defined in Equation 8.6, enables the previous functions to be defined as in Equation 8.7 and 8.8.

$$reduce : (\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}) \rightarrow \mathbb{Z} \rightarrow (\mathbb{Z}^{\mathbb{N}} \rightarrow \mathbb{Z}) \quad (8.6)$$

$$f : \mathbb{Z}^{\mathbb{N}} \rightarrow \mathbb{Z} = reduce \quad (+) \quad 0 \quad (8.7)$$

$$g : \mathbb{Z}^{\mathbb{N}} \rightarrow \mathbb{Z} = reduce \quad (\cdot) \quad 1 \quad (8.8)$$

This is similar to skeleton based programming [82] such as BlockLib [83] and higher order functions in functional programming languages such as F# [84], Haskell [85] and OCaml [86]. For data parallelism on larger

data sets we can of course split the sets over several compute clusters. In general, tasks should be stateless functions following the form

$$f : P_1 \times \dots \times P_n \rightarrow R_1 \times \dots \times R_m. \quad (8.9)$$

Here, the input is some tuple of parameter types and the output is some tuple of result types. This is without loss of generality w.r.t. what can be computed, as we can trivially simulate any state by simply adding a state to the input and output. E.g. assume that we have a procedure/function p that uses state and has the type

$$p : P \rightarrow R. \quad (8.10)$$

This can just be converted to

$$f : P \times S \rightarrow R \times S \quad (8.11)$$

where S is the type of the state.

So why should the program be constructed using functions?

Simplified programming In order to use the resources of ePUMA it takes a lot of work if one must manually issue all data transfers, handle the active memories and so on. For example, if we have a function $f : P \rightarrow R$ that we want to apply to all elements of a buffer, we could use a library function called *map* (a functional) that constructs a new function that operates over a data set. E.g.

$$\text{map}(f) : P[] \rightarrow R[] \quad (8.12)$$

The benefit of this is that the library provider can take care of generating the most suitable schedule for performing the computation.

2

Simplified scheduling It is clearly visible to a scheduling tool which data is still relevant after a task has completed, and the data can be migrated to some other core if it should be suitable.

²For optimal results this requires either profiling of the functions or programmer annotations specifying the expected execution time of the function.

Composition is easy It is very easy to transform small functions into larger functions. E.g.

$$\text{map}(f) \circ \text{map}(g) \equiv \text{map}(f \circ g). \quad (8.13)$$

Without side effects, this transformation is legal and can be automatically applied.

Composition is necessary Assume that we have a task t for a completely data parallel problem. If it takes n cycles to load task t 's input data, and n cycles to store its results, then unless the execution time of t is larger than $2 \cdot n \cdot C$, where C is the number of active CCs, we will be IO bound.

Non-static memories The memories connected to an MPE changes during the execution of a kernel, as we want to hide memory transfers behind computation. To do this we need to select which memories are used for computation and DMA respectively, at any given time. Mapping kernels that do not depend on state is simpler than keeping track of what data is in what memory. In the kernels that we've considered this is not a problem, and, as previously noted, it is easy to represent state as a set of data values that are passed around.

Within the MPEs, a transformation such as $\text{map}(f) \circ \text{map}(g) \rightarrow \text{map}(f \circ g)$ is not as trivial w.r.t. performance, as running g after f is likely to introduce some dependencies. However, given the efficient iteration constructs in the MPEs we might as well just run it as $\text{map}(f) \circ \text{map}(g)$. This leads to more memory accesses and will require slightly more energy. The main issue with resolving this is that the register size is rather small for ePUMA combined with a very long pipeline. However, "normal" compilation is not a topic of particular interest of ePUMA, other than the actual practicality for a production ready system.

8.3 Discussion & Summary

In this chapter a way of programming scratch-pad based systems such as ePUMA has been introduced. While the complexity is much higher than for many conventional systems, it shares many similarities with other high-performance systems. The extra steps required for application programmers should be of a similar complexity.

Chapter 9

Toolchain Support

During the ePUMA project a large set of tools have been developed. In this chapter we will go through the tools implemented and how they relate to the other topics in this thesis.

9.1 Scheduler

One of the issues we're faced with when programming on ePUMA is managing all the memories, data transfers etc. The scheduler is a program that takes a task graph as in Chapter 5, and solves the resulting ILP problem. At this stage it only provides a schedule and is not integrated with the other tools.

Figure 9.1 shows the main flow of the scheduler. The dependency checker analyzes which activities are dependent, in order to reduce the number of constraints. The symmetry checker analyzes the graph and searches for parallel subgraphs with the same shape. It imposes an execution order for activities and on which resource, e.g. core, that should be used. The constraint builder adds constraints, as in Chapter 5, and uses an external tool for solving the problem and providing a schedule. It is implemented in F# [84] and can use either Gurobi [37] or Google OrTools [38] as an ILP solver.

Figure 9.1: Scheduler program flow.

9.2 Memory Resolver

The memory resolver is tool for allocating logical memory addresses to physical addresses in such a way that memory bank conflicts are minimized. It is based on the methods described in Chapter 6. It can either take a set of accesses as input and permute the addresses as in Section 6.3, or sequences as and use padding as in Section 6.4. The memory resolver can function as a standalone tool, but is also included in the assembler and FT compiler.

The flow for one buffer is illustrated in Figure 9.2. If the input access pattern is a list of vector accesses the method describes in Section 6.3, permute, is used. First, the accesses are converted into a graph giving all the connected components. As many of the graphs are isomorphic, the graphs are converted to a unique set of graphs. These graphs are then solved individually by using a constraint builder, as in Section 6.3, and feeding the constraints into an ILP solver. After each address has been

assigned a memory bank, a post-processing step creates the new address mapping.

Figure 9.2: Memory resolver flow.

9.3 FT Compiler

FT is the domain specific language for writing computation kernels for ePUMA. FT is a smaller abbreviation of Formula Translation than Fortran is. FT shares many design goals with Fortran but with a much smaller application scope. It is based around vector operations and solves the more problematic parts of writing MPE code. E.g. allocating data buffers to memory, using address registers in an optimal way etc. It is implemented similarly to other compilers and F# is used as an implementation language. It can resolve bank conflicts by using the memory resolver. A brief introduction to the other features provided is given in Section 7.1.

Currently FT only compiles programs that adhere to the assumptions of ePUMA's application domain, since the effort of making it useful for

the general case is not justifiable for the research purposes of this project.

There is currently no specification of FT, other than its implementation, and as it is not an exercise in language design. The next step if this project is taken any further would be to migrate the optimization passes to some more well known compiler framework, meaning that a different front-end language, but with similar capabilities, would be used.

The flow for the FT compiler is shown in Figure 9.3. It takes a FT source file and generates an assembly file.

Figure 9.3: FT flow.

9.3.1 Data placement

A major part of the FT compiler relates to data placement.

Memory resolver The memory resolver tools is used to resolve memory conflicts for accessing different buffers.

Memory allocation The memory allocation stage uses an ILP model as described in Section 7.2.1. In the same way as the other tools are implemented, a constraint builder creates a model that is solved by an external tool.

Address allocation This stage allocates the where in the memory the buffers should reside, by using simple overlap constraints.

Register allocation Identical to the address allocation stage, except for variables that should be allocated to the register file.

Address register allocation This stage assigns which address register an instruction should use when traversing a data structure. This is just an assignment based on which register was used least recently in the Intermediate Representation (IR). A more clever strategy have not been necessary for the kernels that we have evaluated so far.

9.3.2 Other Stages

In order to generate decent code for the MPEs, a few other stages have been implemented as well. Some are quite standard compiler passes while some are more ePUMA specific.

Tokenizer/parser A standard tokenization/parsing stage.

Semantic analysis/type checking Currently the analysis here is rather limited. However, some basic checks are performed, e.g. that different operations have compatible shapes.

Pattern analysis This stage simply performs a simple pattern analysis, e.g. replacing a summation of an element-wise vector multiplication with a MAC node.

Stride application Performs the optimization described in Section 7.3.

Peephole optimization Performs some simple iteration based optimizations. E.g. if an inner loop have been reduced to a single instruction that is enclosed in a repeat loop, the loop is replaced with the same instruction except with the instruction count multiplied with the loop counter.

Assembly emitter This step takes the resulting IR and generates an assembly file.

9.4 mpeasm

mpeasm is the high-level assembler for the MPEs. In a sense it is more like a compiler than a traditional assembler, as there are many transformations necessary to generate the final binary. It provides an extensive set of semantic checks, can resolve memory conflicts, and eliminate duplicate data from the binary, such as duplicate permutation vectors or constants. It was originally developed in OCaml but is currently implemented in F#, and uses the memory resolver for conflict-resolution.

The flow for *mpeasm* is shown in Figure 9.4. It is implemented more or less as a compiler due to the complexity (from a software point of view) of the target hardware. Most of the steps quite normal, e.g. the tokenizer and parsing steps. The intermediate code generation step calculates e.g. labels and other necessary constants, while the code generator emits the binary code.

9.4.1 Assembler Inputs

The assembler takes two inputs. During build time (of the assembler) an instruction specification must be given, and when used for assembly an assembly file must be provided.

Instruction specifications The instructions available and their formats, e.g. valid operand combinations for the instructions are quite nu-

Figure 9.4: mpeasm flow.

merous, and it is not feasible to rewrite the assembler for each potential hardware revision with different instruction sets and hardware resources (e.g. available memories). These hardware specifications are read by the assembler and is used when verifying that the assembly code is valid, and for generating the final binary code.

Assembly file An assembly file written according to the ePUMA programming manual.

9.4.2 Assembler Outputs

The assembler generates three files as output.

Object file A binary object file that can be processed by the linker and

executed by the simulator.

Header file A file that can be included into a C program so that the kernel can be called.

Mapping file A file that maps addresses to source lines, in order to facilitate debugging when using the simulator.

9.4.3 Semantic analysis

Due to the complexity of the MPE hardware, it became necessary to include a stage that can emit somewhat sensible error messages for invalid assembly programs. A short, but incomplete, list of examples follows:

Instruction format validation The different operand modes available varies for different instructions. E.g., butterfly instructions cannot be used with scalar operands.

Control instruction sequence validation The MPEs have different control instructions for e.g. jump, data dependency fences, etc. These are available as “assembly instructions”, but are actually packed into special instructions where some combinations of control instructions are not available due to lack of bits in the control field for in the opcode.

Memory usage validation A common mistake is to use a memory multiple times in an instruction, forgetting that the data should be split over several ports. The semantic analysis ensures that there are no hardware conflicts for any instruction.

9.4.4 Optimization

mpeasm will perform the following optimizations:

Control code compression This stage is used to pack several control code instructions into one.

PM data optimization Much of the data stored in program memory relates to e.g. permutation tables, which often contain duplicates. A common case when this occurs is when smaller kernels are merged to form larger kernels and there are common permutation patterns or common data. This stage removes duplicate data segments. It also checks whether a data segment is a subset of another data segment.

Repeat/jump/call optimization When a jump of any kind is performed on the MPEs, one clock cycle may be wasted if the target instruction is not aligned, as the program memory isn't a multi-bank memory. This stage adds an *empty* (or removes a *nop*) instruction before any repeat loop if the first instruction in the body is not aligned properly. An *empty* instruction takes 0 cycles to execute.

9.5 linker/loader

The linker/loader for the project is implemented in C++, as the loader part needed to be usable from other software. In this particular case the system and MPE simulators are written in C++.

Figure 9.5 shows how this tool works. There can be several APE/CPE/MPE object files, but for simplicity only one instantiation of each is shown. This tool can both pack/unpack any file as indicated by the double-ended arrows.

Figure 9.5: Linker/loader flow.

9.6 Data Access Template Configurator

This tool was created to be able to write generic data access code, comparable with higher-order functions in functional languages. It works by allowing assembly code to have annotations that e.g. specifies where code segments should be inserted. This tool has mostly been superseded by *mpeasm*. The flow of this tool is shown in Figure 9.6. The input is an assembly program, code segments that should be inserted into the code, and other parameters. The tool parses the file, and allows for simple directives comparable to the C pre-processor. E.g. if a parameter datatype is set to complex, a complex multiplication instruction *cmul* could replace a normal multiplication *mul*. It also allows for code parameters that can be expanded into the assembly program. The intermediate representation is then converted into a normal assembly program by the assembly emitter.

Figure 9.6: Template configurator flow.

9.7 Configurator

The *Configurator* tool is used for automatically selecting the most appropriate templates depending on the arguments when calling a kernel from application code. E.g. switching algorithms depending on the size of input etc. The purpose of this is that the application code should be as portable as possible over hardware configurations. E.g. the number of CCs could be different or the SIMD width, which could change for which input sizes certain algorithms are suitable. Unfortunately it has not been used that much due to a limited kernel library, and the usefulness of such a tool is questionable. The flow for this tool is shown in Figure 9.7. The input is a call specification, with e.g. parameters such as a function (e.g. sort, FFT etc.), input buffer size etc. It also needs a specification for each template for which range of parameter values it is valid. E.g. a template could specify that the input buffer size should be between 64 and 128. All of these constraints are used as the input to Facile [87], a constraint solving library for OCaml. Facile was used simply because the

solver performance is irrelevant and rest of the configurator is written in OCaml. As there could be a set of templates that fulfill the requirements, the cycle times for executing for each valid template is compared in a template selection stage for the specific parameters, and the fastest kernel is chosen.

Figure 9.7: Configurator flow.

9.8 Toolchain flow

The flow between the different tools is shown in Figure 9.8. Given a kernel call the configurator tool chooses the most appropriate kernel from the kernel library. Depending on the type of kernel it is either scheduled with the scheduler or uses a predefined schedule. When a kernel is scheduled, the code can be compiled or assembled and we finally receive an object file. Absent from Figure 9.8 is the linker/loader, as it is only used after all the kernels for a program have been instantiated.

Figure 9.8: Toolchain flow.

Part IV

Results and Conclusions

Chapter 10

Evaluation

When evaluating the resulting performance by applying the different methods described in this thesis we will use two different metrics aside from just a cycle count.

The first one is utilization. We define utilization as

$$\text{Utilization} = \frac{\text{Computational cycles}}{\text{Total cycles}}. \quad (10.1)$$

Computational cycles are the clock cycles where we issue an instruction that performs an instruction necessary for computing the result, i.e. not an instruction related to controlling the computation. This is a measure of how efficiently we are using the processor, assuming that we're using the most appropriate instructions for execution. I.e., it is pretty trivial to achieve full utilization if we would just be running scalar code, as we would never get bank conflicts for example.

In addition to utilization we will also use relative speedup as a metric, defined as:

$$\text{Speedup} = \frac{\text{Total cycles the slower version}}{\text{Total cycles for faster version}}. \quad (10.2)$$

It is very easy to manipulate this metric (obtain a large value) by comparing a well-optimized fast version with a very poor baseline implementation. In our measurements, we will use the same implementation strategy and algorithm to implement both the fast and slow versions. The

machine instructions for performing the actual computation will therefore be the same. The purpose is instead to measure the performance impact of a single technique, such as a compiler optimization. Relative speedup is then a good way to measure the improvement.

10.1 Permutation

In this section we will look at the performance improvements when using permutation for selected problems. The performance results for the solution of the ILP problems were obtained on a computer with 24 GB of RAM and a 3.46 GHz six-core processor.

10.1.1 Preprocessing Performance

Finding conflict-free solutions can take time if the input data set is large. For example, if we have $N \ 8 \times 8$ blocks with row and column access, feeding all of them into the solver will take some time, as shown in Figure 10.1. Obviously, if the access pattern within each block is the same for all blocks, we could just solve the problem for one block and use the obtained permutation for all blocks. This requires recognizing that the blocks can be separated. This could possibly be done by analyzing the loops within a kernel and try to determine whether the accesses operate on blocks in an identical way. However, as we are already operating under the assumption of static addressing, we can simply generate an undirected graph where addresses that are accessed at the same time are connected. This can be reduced by considering each connected component by itself as described in Section 6.3.2. By doing this, the solution time is reduced considerably, as shown in Figure 10.2. It should be noted that this is a desirable thing to do even if we do not consider the solution times, as we want to get the same permutation vectors for identical components.

In media applications it is common to access data in different zig-zag patterns during compression/decompression. Figure 10.3 shows the access cost as function of time for a collection of patterns. The patterns

Figure 10.1: Solution time without preprocessing.

include row, column and the zig-zag pattern on a data block. Two of the patterns are from the H.262 standard [88], and one from the JPEG standard. In many cases it is possible to find alternatives to permutation by using e.g. more memory, as in Section 6.4, but for zig-zag patterns there really aren't really any alternatives.

Table 10.1 shows the solution times to find an optimal solution and to prove that it is an optimal solution for the different access patterns. The table also the average access costs for accessing the data blocks. R denotes row access, C denotes column access, Z_x denotes a zig-zag pattern. $X \cup Y$ denotes that the patterns X and Y will be used to access a block. It is difficult to predict the solution times as they can vary by orders of magnitude. However, for the zig-zag 2 case (third entry) which is slow to solve and does not yield a conflict-free solution, we can see that if we split the data buffer into two buffers (last entry), the solution time is very fast. This is a good motivation for adapting a more functional programming style and not mutate buffers, as it decreases the access pat-

Figure 10.2: Solution time with preprocessing.

terns per buffer, and makes the problem much easier. Also, if we split the buffers, then we know that we can find a conflict-free solution, and use a specialized algorithm for it.

Table 10.1: Solution times and average access costs.

Access pattern	Solution time in seconds	Average cost
$R \cup C$	0.0156	1.00
$R \cup C \cup Z_1$	3.5880	1.00
$R \cup C \cup Z_2$	456.7376	1.08
$R \cup C \cup Z_{jpeg}$	9.1104	1.00
$R \cup C \& C \cup Z_2$	0.0312	1.00

Figure 10.3: Solution time for ZigZag patterns.

10.1.2 M-JPEG

Next, we will evaluate the effect of permutation on part of the M-JPEG encoding/decoding process. All computing steps necessary for M-JPEG have been included, except for Huffman encoding/decoding, which in our implementation has been offloaded to a hardware accelerator. Results for a full implementation on a FPGA prototype of ePUMA can be found in [89], where an M-JPEG encoder compresses data with resolution 1920×1080 at 30 fps, using a configuration with 4 MPE cores.

M-JPEG encoding can be implemented, without permutation, as shown in Figure 10.4. We must transpose the output from the first DCT before executing the second so that the 2D-DCT is correctly computed. We then quantize the values and reorder them to be consumed by the Huffman encoder. With permutation, we only need to do the 1D-DCTs and quantization, as in Figure 10.5, as then we can access the relevant data without any separate reordering process.

Figure 10.4: Part of M-JPEG encoder using a shufb equivalent instruction.

Figure 10.5: Part of M-JPEG encoder using permutation.

The M-JPEG decoding process is easier both with and without permutation. Without permutation, we can more or less perform the encoding process in reverse. However, since the MPE provides an instruction which performs multiplication followed by 1D-DCT, these can be merged into a single step, as in Figure 10.6. This just leaves the cost of

Figure 10.6: Part of M-JPEG decoder using a shuffle equivalent instruction.

1D-IDCTs when using permutation, as in Figure 10.7.

Figure 10.7: Part of M-JPEG decoder using permutation.

In Table 10.2 we can see the cycle times per block. (I)ZZT denotes (Inverse) Zig-Zag with Transpose. Without permutation, quite a lot of time is spent just reordering data, so for this particular case permutation is obviously useful. In Table 10.3 we have the total execution time for one

Table 10.2: Execution times in clock cycles per block on one MPE core.

Algorithm	With shuffling	With permutation
1D-DCT	16	16
1D-IDCT	16	16
Transpose	24	0
Q	8	8
IQ	0	0
ZZT	32	0
IZZT	32	0

block. As the shuffling of the zig-zag pattern could be implemented in the Huffman encoder/decoder, we also include the case when we shuffle and skip the zig-zag step. Either way, the solution with permutation is

Table 10.3: Total execution times in clock cycles per block on one MPE core.

Algorithm	shuffling	shuffling without ZZ	permutation
Encode	96	64	40
Decode	88	56	32

significantly faster, and we have the speedup in Table 10.4.

Table 10.4: Speedup by using permutation.

Algorithm	$\frac{\text{shuffling}}{\text{permutation}}$	$\frac{\text{shuffling without ZZ}}{\text{permutation}}$
Encode	2.40	1.60
Decode	2.75	1.75

10.1.3 FFT

One algorithm that benefits greatly from permutation is the FFT. Figure 10.8 shows the time needed to find conflict-free solutions for differently sized FFTs. The FFTs work on complex data, and the effective SIMD

Figure 10.8: Solution times for differently sized FFTs.

width is then 4. Without permutation we get a problem either with writing or reading the data buffers. The result of this is that each butterfly will take four clock cycles instead of one. Table 10.5 shows the execution times for FFTs with and without permutation. This includes all code necessary to compute the FFTs. We can see the speedup in Figure 10.9. The

Table 10.5: Execution times in clock cycles for FFTs on one MPE core.

# points	With permutation	Without permutation
16	47	71
32	83	155
64	116	260
128	245	629
256	416	1184
512	940	2860
1024	1729	5569
2048	4045	13261
4096	7670	26102

speedup for larger FFTs reaches about 3.4. The reason that we don't get a speedup of four is due to loading the twiddle factors.

Returning to Table 6.2, recreated for convenience in Table 10.6, we can see that the cycle times when using shuffling for permutation are quite large compared to the total execution time for an FFT on a MPE. E.g. a 256 point shuffle, for one layer, is 106 cycles. A complete 256 point FFT on an MPE is 416 cycles.

Table 10.6: Number of shuffling instructions.

Stride	Size	Cycles for Core2 SSE2	Cycles for Cell SPU
2	4	4	15
4	16	13	22
8	64	35	
16	256	106	

If we for simplicity assume that we can use the cycle time for the

Figure 10.9: Speedup by using permutation.

largest permutation for each buffer, i.e. a lower bound on shuffling cost, when computing an FFT and adding this to the cycle time for an FFT on an MPE using permutation we get the results in Table 10.7. Even if we would implement the same method as in [54] we would at best get marginally better result than our implementation without permutation.

Table 10.7: Comparison between permutation and shuffling.

# points	With permutation	With shuffling	Speedup
16	47	73	1.55
64	116	221	1.91
256	416	840	2.02

10.2 Padding

In this section we will consider the effects that padding has on sorting and FFTs.

10.2.1 Sorting

The sorting algorithm used here is bitonic sort. The MPEs have specific instructions to help with sorting vectors and merging, and ideally processes one vector per clock cycle.

We can see access speedup in Figure 10.10. For a small number of elements the strides do not become that large, and we will quite naturally, as discussed in Section 6.4, have few memory bank conflicts. For larger vectors, i.e. the factors contain more 2s, we see an increased effect.

Figure 10.10: Access speedup for sorting.

In Figure 10.11 we can see the memory overhead from padding when sorting. It levels off at 12.5%, as indicated by Equation (6.43).

We also need to store the permutation vectors and we can see the

Figure 10.11: Memory overhead of sorting.

effect padding have on those. It should be noted that we would need most of these anyway as we still need the vectors to specify the addresses in the unpadding case.

The execution time for sorting is in Table 10.8. The standard way is without a multi-bank memory at all, i.e. only consecutive elements may be fetched. Multi-bank means that we use permutation vectors but do not pad the buffer, and padding of course means using padding.

In Figure 10.13 we have the speedup by only using multi-bank memories. The multi-bank feature does in itself not provide that much speedup for larger problems. However, if we add padding, we get quite significant speedups, up to almost a factor of 8 for larger problems as shown in Figure 10.14.

Figure 10.12: Number of permutation vectors necessary for sorting.

Table 10.8: Execution times in clock cycles for sorting on one MPE core.

# elements	Padding	Multi-bank	Standard
16	36	40	99
32	66	98	234
64	120	296	568
128	253	861	1485
256	546	2402	3682
512	1207	6711	8823
1024	2835	17683	21651
2048	6575	44975	52271
4096	15051	113355	120779

Figure 10.13: Speedup for sorting with multi-bank hardware without padding.

Figure 10.14: Speedup for sorting with padding.

10.2.2 FFTs

For FFTs the memory overhead will be larger as we use complex data. We will use the same metrics as for sorting, except we will also consider the result when we constrain the maximum amount of memory used. That is, the evaluation is performed both with unrestrained memory usage and by applying a 10% constraint on additional memory required.

We have the average access costs in Figure 10.15. Without memory constraints we get conflict-free access.

Figure 10.15: Average access cost for FFT.

The speedups for the FFTs are shown in Figure 10.16. The access is at least two and approaches 3.5 for larger FFTs. As we use complex data, by the same reasoning as in Section 10.1.3, we can conclude that the speedup should be below four.

The memory costs for padding FFTs are shown in Figure 10.17. By the same reasoning that leads to Equation (6.43), 25% seems quite reasonable.

The number of permutation vectors are shown in Figure 10.18. As for

Figure 10.16: Access speedup for FFT.

Figure 10.17: Memory overhead for FFT.

sorting, we would need these vectors anyway.

Figure 10.18: Number of necessary permutation vectors.

We have the execution times for differently sized FFTs in Table 10.9, and the speedup is shown in Figure 10.19. As is the case if we use permutation, the speedup approaches 3.4.

Table 10.9: Execution times in clock cycles for FFTs on one MPE core.

# points	Padding	Standard
16	47	71
32	83	155
64	116	260
128	245	629
256	416	1184
512	940	2860
1024	1729	5569
2048	4045	13261
4096	7670	26102

Figure 10.19: Speedup by using padding for FFTs.

10.2.3 FFTs including Non-power 2 FFTs

The values for differently sized FFTs with the addressing features of the latest revision of the MPEs, as well as with the optimizations in Chapter 7, are shown in Table 10.10 and Table 10.11. Unfortunately we only have values with conflict-free addressing and the minimal number of instructions.

Table 10.10: Execution times in clock cycles for FFTs on one MPE core.

# points	Minimum	Execution time	Utilization
256	256	322	0.795
288	792	872	0.908
300	1215	1281	0.948
324	1377	1457	0.945
360	1404	1484	0.946
384	768	848	0.906
432	1512	1592	0.950
480	1512	1592	0.950
512	640	720	0.889
540	2511	2591	0.969
576	1584	1664	0.952
600	2580	2660	0.970
648	2916	3010	0.969
720	2808	2888	0.972

The absolute execution time is shown in Figure 10.20, and the utilization is shown in Figure 10.21. It can be noted that mixed radix FFTs have a higher utilization than FFTs of a power of 2. The reason for this is that radix-3 and radix-5 butterflies does not map as well to the data path, and

Table 10.11: Execution times in clock cycles for FFTs on one MPE core.

# points	Minimum	Execution time	Utilization
768	1536	1616	0.950
864	3240	3334	0.972
900	4545	4625	0.983
960	3024	3104	0.974
972	5103	5197	0.982
1024	1280	1360	0.941
1080	5292	5386	0.983
1152	3456	3550	0.974
1200	5160	5240	0.985
1296	5832	5926	0.984
1536	3456	3550	0.974
2048	3072	3166	0.970
3072	6912	7006	0.987
4096	6144	6238	0.985

cannot be implemented as a one cycle instructions, thus illustrating the danger with utilization as a metric.

Figure 10.20: Execution time for mixed radix FFT implementation.

Figure 10.21: Utilization for mixed radix FFT implementation.

10.3 Vector Code Compilation

In this section we will evaluate the results when compiling vector based code. We will consider some algorithms with common structures.

10.3.1 FIR filter

As a first example we will consider an FIR filter/1D convolution that is quite typical in many DSP applications. The code in FT is shown in Listing 10.1.

Figure 10.22 and Figure 10.23 show the execution time in clock cycles for the non-optimized and optimized version respectively. The number of data points (n) is 1024. The overhead is a rather large part of the execution time for the non-optimized version, while it is hardly noticeable in the optimized case.

We can see the utilization in Figure 10.24. With the optimized version we get almost full utilization. For small vectors the overhead without optimization completely dominate the computation time, and here it is obviously beneficial to apply it. We can also see that the gap closes quite slowly, so for them to be comparable requires very large filters.

The speedup is shown in Figure 10.25. For small vectors the speedup is around 7, which is quite extreme. Usually the filters are a bit larger, but even with as many as 70 taps the optimized version is still twice as fast.

Listing 10.1: FIR filter

```
1 for i in 0 .. n do
2     ys[i] = reduce (+) (xs[i..m] * coeffs);
3 done
```

Figure 10.22: Execution time in clock cycles for the non-optimized version.

Figure 10.23: Execution time in clock cycles for the optimized version.

Figure 10.24: Hardware utilization for a FIR filter.

Figure 10.25: Speedup for FIR filter.

10.3.2 Matrix multiplication

Next, we will consider matrix multiplication. We're using a quite simple implementation, as shown in Listing 10.2, where we calculate each element as the scalar product of two vectors.

Figure 10.26 and Figure 10.27 shows the execution time in clock cycles for the non-optimized and optimized version respectively. The overhead is a rather large part of the execution time for the non-optimized version on smaller matrices, while it is hardly noticeable for the optimized case.

The utilization is shown in Figure 10.28. For smaller matrices the utilization is just above 0.7. However, it reaches almost full utilization quite fast, and the effect of the optimization is substantial. The speedup is shown in Figure 10.29. We see the same pattern as for the FIR filter, for smaller problems we have a significant speedup, in this case over 2.5, but we tend to one as the sizes increase.

Listing 10.2: Matrix multiplication

```
1  for i in 0 .. n-1 do
2      for j in 0 .. m-1 do
3          zs[i,j] = reduce (+) (xs[i,:] * ys[:,j]);
4      done
5  done
```

Figure 10.26: Execution time in clock cycles for the non-optimized version.

Figure 10.27: Execution time in clock cycles for the optimized version.

Figure 10.28: Hardware utilization for matrix multiplication.

Figure 10.29: Speedup for matrix multiplication.

10.3.3 Multiple matrix multiplications

It is quite common to perform many small matrix multiplications. Figure 10.30 and Figure 10.31 shows the execution time in clock cycles for the non-optimized and optimized version respectively when multiplying n 8×8 matrices. The overhead is a rather large part of the execution time for the non-optimized version, while it is significantly reduced for the optimized case.

Figure 10.30: Execution time in clock cycles for the non-optimized version.

As the utilization is quite poor when performing one single matrix multiplication if the matrices are small, we still get a high utilization when performing many multiplications, as shown in Figure 10.32. In this case, the speedup is pretty constant, stabilizing just under a factor of 2.5, as shown in Figure 10.33. This is quite natural as the vector instructions are only issued on small vectors, and the slower version must continually use some overhead when changing matrices.

Figure 10.31: Execution time in clock cycles for the optimized version.

Figure 10.32: Hardware utilization for multiple matrix multiplications.

Figure 10.33: Speedup for multiple matrix multiplications.

10.3.4 2D convolution

For 2D convolution the implementation is required to be a bit more complicated than the natural way of implementing the algorithm. This is because for efficient execution we need to use the vector accumulator register, which is of limited width equal to the datapath width. The implementation is shown in Listing 10.3, where we accumulate vectors of length W .

Figure 10.34 and Figure 10.35 shows the execution time in clock cycles for the non-optimized and optimized version respectively. The overhead is a significant part of the execution time for the non-optimized version. For the optimized case it is hardly noticeable.

Figure 10.34: Execution time in clock cycles for the non-optimized version.

Figure 10.36 shows the utilization for 2D convolution. The utilization is quite high even for very few non-zero coefficients, over 0.8, and eventually approaches one.

For small kernels the speedup is up to a factor 4, and approaches one

Listing 10.3: 2D convolution

```

1  for row in 0 .. rows-1 do
2      for col in 0 .. W .. cols-W do
3          acc = zeros();
4          for coeff in 0 .. n_coeffs-1 do
5              acc += xs[offsets[coeff] + row,col:W]
6                  * coeffs[coeff];
7          done
8          ys[row,col:W] = acc;
9      done
10 done

```

Figure 10.35: Execution time in clock cycles for the optimized version.

with larger problem sizes. While one might think that given the short vectors that are accumulated the speedup would remain high as for the multiple matrix multiplications, the reason here is that the addressing mode generally used for these scenarios is recognized and the inner loop is executed as an instruction with an iteration count, i.e. there is no loop-

Figure 10.36: Hardware utilization for 2D convolution.

ing overhead.

Figure 10.37: Speedup for 2D convolution.

10.4 LTE Downlink

A full system implementation, i.e. an implementation that uses several compute clusters, of an LTE downlink chain was implemented and is described in [17]. The final performance and utilization for the MPEs is shown in Table 10.12. MPE 0 and 1 are used to compute the FFTs for

Table 10.12: Execution times in clock cycles for FFTs on one MPE core.

Compute cluster	Computing cycles	Total cycles	Utilization
0,1	9578	13902	0.689
2	2300	2751	0.836
3	28500	32351	0.881

two receiving antennas, while MPE 2 is used for channel estimation, and MPE 3 is used for channel equalization. The problem with the FFTs in

this case is that we have too few memories, and run out of memory ports due to the DMA transfers. This makes an argument for either more local memories, or perhaps to use dual port memories instead.

The DMA overhead for this application is 5692 clock cycles, or 10.1% of the total, 56488 cycles.

Unfortunately the rest of the ePUMA system is not capable of achieving the same utilization as the MPEs; however, it is still quite high.

10.5 Summary

This chapter presented the results of applying software tools to solve data layout problems, and optimizing the execution of vector based code by applying 2D addressing. While the set of algorithms are limited they do represent the class of target algorithms rather well.

We have shown that we can solve the data layout problems automatically, and that the permutation network can be utilized without incurring any software development costs.

We have also shown that we can achieve very high utilization on the MPEs, at least as long as we have an IR that allows us to use vector operations. From a software point of view, utilizing the MPEs effectively is not a problem, as hand coded solutions are not more efficient. I.e. not much more can be done from the software side.

While the results for the full system has not yet proven quite as encouraging, it is a hard problem to solve as high throughput off-chip is very expensive, so there is a limit to how well it can be done.

Chapter 11

Conclusions and Future Work

The purpose of the ePUMA project was to design a DSP platform that would be suitable for applications with predictable data access. While the design phase for the platform itself is nearing completion, the major part lacking is the evaluation by implementation of actual applications and benchmarking them. Ideally, we would also like to tapeout a test chip so that we could measure power consumption, final frequency, have hard area figures etc.

11.1 Programming Model

The programming model used in ePUMA is not that different from mainstream software development. As such, ePUMA will not incur any major extra costs for software development.

11.2 Conflict-free Data Access

For static access patterns, which was the original goal, we have shown that it is quite possible to automate the process and removing most of the effort required by programmers. The question is of course if these

features are common enough to warrant the hardware cost. At least in some examples the performance difference is quite big; but, even for those cases, if we look on the overall performance of an application, will it be significant enough to matter? If we often end up to be I/O bound one might even consider the MPEs to be unnecessarily fast.

11.3 MPE compilation

Programming in assembly is largely agreed upon to be a primitive way of developing software, and it only worsens with more complicated features available in the processors. For the MPEs, getting the addressing right is the primary issue. However, using this efficiently from a higher level language is not an issue, given that the problem is from the intended application domain at least. However, for the MPEs to be able to be used from a practical point of view, the FT compiler should be extended to handle more general code even if those code segments can't be compiled into efficient code. Or rather, the optimization passes in FT should be ported to another compiler framework.

11.4 Summary

Given that the final hardware will be able to run with a high enough frequency at a reasonable power consumption, and that it will have a small enough area, there is nothing stopping ePUMA from a software development point of view. While it does contain some features that are perhaps a bit unusual, it is possible to move the complexity to software tools.

11.5 Future Work

The most obvious thing to continue with is to validate the design by mapping more applications to ePUMA. This is important not only for

benchmarking the processor, but also to know whether or not our initial assumptions actually still hold for modern applications.

Currently the MPEs are programmed in assembly or FT. However, this requires porting of programs if we wish to run them on the MPEs. It would be interesting to take FT and convert it into a back-end for a compiler framework such as LLVM or GCC, and see if it is possible to compile code written in more conventional languages. This would alleviate some of the burden of application development, and also determine if the ePUMA design is viable in a commercial setting.

It would also be interesting to see if the MPEs or some adaptation could be used in a context outside of ePUMA. As the HPC community becomes more and more interested in power efficient solutions, and that the MPEs are very good vector processors, it is possible that the MPEs could be a good fit as co-processors in larger HPC clusters.

Chapter 12

Retrospectives

In this chapter we will revisit the process of the ePUMA project, what we should have done differently, what we did right, and some general thoughts.

12.1 Scope

The point of any research is to be able to tell something about the world. The purpose of ePUMA was to investigate whether or not it is possible to design an ASIP DSP platform usable for a large set of applications with predictable data access, in particular embedded applications, and how usable it would be. In particular, if is it possible to reach the limit of the available memory bandwidth.

What is necessary to fully answer this question? We would need to know the areas of different configurations necessary to run on each platform, the power consumption when running the applications as well as the performance. In other words, it is necessary to implement the platform, all software tools necessary to program it, and a bunch of real world applications.

The problem with this is that this is quite a lot to do for just a few Ph.D. students. Considering that many projects use established platforms and still take quite a while to complete, it might have been better to have some focused on more specific issues that can be answered with

a bit more reasonable effort. As we haven't been able to really provide a complete answer the original problem statement, our contribution is still on the level of sub goals.

We have shown that at least for some cases we can more or less fully utilize the bandwidth available for the MPEs, and the MPE hardware design and software tools seem to have met the design goals. If everything goes as planned, final hardware figures and a wider array of kernels, to substantiate this claim further, will be available by the end of the year (2015) in Andréas Karlssons thesis.

Unfortunately, answering whether the full ePUMA platform, i.e. everything outside the compute clusters, will fulfill the initial design goals seems to be a bit more difficult and unlikely to happen. While we had a great opportunity to create a spin-off company, some practical rather than technical issues made it fall through. This is a shame, as it would have provided the resources to fully verify that the ePUMA platform is suitable for future DSP.

12.2 Number of hardware iterations

There has been too many iterations of ePUMA and the MPE designs. While it is hard to avoid this in a large project, we should have gone about this in different way, as many of them were quite unnecessary.

The first iteration of the MPEs for example was only designed with hardware cost and performance in mind. This can in a way make sense, as the performance and hardware cost must be excellent for the project to be considered a success. On the other hand, while a large corporation might have the resources to have someone implement a lot of code in assembly, and in this case very complicated assembly, as small team of Ph. D. students don't. As the hardware was quite specific, developing a compiler was also difficult as there is no compiler framework that really fits. While many issues with the hardware could have been solved, the sheer amount of issues would make the effort necessary to complete a full compiler to simply be unreasonable.

The error we made is that we focused too much on what computational kernels that different applications used, and focused the design on this. While this may lead to an efficient design, all the other code still has to be written; and, for e.g. video codecs, this can be a substantial amount if existing code cannot be reused in a meaningful way.

The latest version of ePUMA is programmable in a quite conventional way, and it is now in a state where one might consider to start porting software to it, as we would mostly have to rewrite some kernel parts. However, with a bit of thought and analysis of the work required, we should have been able to realize that we should have started in this end.

12.3 Order of work

Normally projects have different phases that require different competences and kinds of work done. While knowledge about compilers etc. is important when designing a processor, and it can be useful to discuss these issues in this phase, it is difficult to start working on software tools in this phase. Without a specification of the hardware, it is all but impossible to create something that can generate code for the platform. When a specification by implementation does exist, continued hardware changes requires rework of the software tools as well. This is on the other hand probably hard to avoid, as we're just designing one system, contrary to companies that have a pipeline of products coming out and allows for people to move between projects, and uses iteration over products to a larger extent than is possible here.

Secondly, it makes it difficult to publish papers from a software point of view, as many problems are ePUMA specific, and the only point of solving them is if the ePUMA hardware platform itself is useful. On the other hand, they need to be solved to be able to use the hardware platform.

12.4 Final remarks

All problems aside, I believe that the things we've learned during this project is far more substantial than if we wouldn't have built our platform from the ground up. Dealing with all the details have provided insights that would have been hard to acquire by other means. So while we could perhaps have done it better w.r.t. to what results we ended up with, and with less effort, from an educational point of view it has been a very valuable experience.

Bibliography

- [1] Dake Liu, A. Karlsson, J. Sohl, Jian Wang, M. Pettersson, and Wenbiao Zhou. ePUMA embedded Parallel DSP processor with Unique Memory Access. In *Information, Communications and Signal Processing (ICICS) 2011 8th International Conference on*, pages 1–5, dec. 2011.
- [2] C. H. (Kees) van Berkel. Multi-core for mobile phones. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE '09*, pages 1260–1265, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.
- [3] ePUMA Team. <http://www.da.isy.liu.se/research/ePUMA/>.
- [4] Amin Shafiee Sarvestani, Erik Hansson, and Christoph Kessler. Extensible Recognition of Algorithmic Patterns in DSP Programs for Automatic Parallelization. *Int. J. Parallel Program.*, 41(6):806–824, December 2013.
- [5] Michael Gschwind. The cell broadband engine: exploiting multiple levels of parallelism in a chip multiprocessor. *Int. J. Parallel Program.*, 35:233–262, June 2007.
- [6] Joseph A. Fisher. Very Long Instruction Word Architectures and the ELI-512. *SIGARCH Comput. Archit. News*, 11(3):140–150, June 1983.
- [7] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings*, 7:483–485, 1967.

- [8] Dake Liu. *Embedded DSP Processor Design: Application Specific Instruction Set Processors*. Morgan Kaufmann, May 2008.
- [9] Dake Liu, Joar Sohl, and Jian Wang. Parallel Programming and Its Architectures Based on Data Access Separated Algorithm Kernels. *IJERTCS*, 1(1):64–85, 2010.
- [10] High Performance Embedded Computing. <http://hipec.cs.lth.se/>.
- [11] Mark Woh, Sangwon Seo, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Krisztian Flautner. AnySP: Anytime Anywhere Anyway Signal Processing. *SIGARCH Comput. Archit. News*, 37(3):128–139, June 2009.
- [12] Hoseok Chang, Junho Cho, and Wonyong Sung. Compiler-Based Performance Evaluation of an SIMD Processor with a Multi-Bank Memory Unit. *Journal of Signal Processing Systems*, 56:249–260, 2009. 10.1007/s11265-008-0229-z.
- [13] Rizwan Asghar. *Flexible Interleaving Sub-systems for FEC in Baseband Processors*. PhD thesis, Linköping University, Computer Engineering, The Institute of Technology, 2010.
- [14] Yuan Lin, Hyunseok Lee, Mark Woh, Yoav Harel, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Krisztian Flautner. SODA: A Low-power Architecture For Software Radio. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture, ISCA '06*, pages 89–101, Washington, DC, USA, 2006. IEEE Computer Society.
- [15] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding Sources of Inefficiency in General-purpose Chips. *SIGARCH Comput. Archit. News*, 38(3):37–47, June 2010.

- [16] Kees van Berkel, Frank Heinle, Patrick P. E. Meuwissen, Kees Moerman, and Matthias Weiss. Vector Processing As an Enabler for Software-defined Radio in Handheld Devices. *EURASIP J. Appl. Signal Process.*, 2005:2613–2625, January 2005.
- [17] Jian Wang. *Low Overhead Memory Subsystem Design for a Multicore Parallel DSP Processor*. PhD thesis, Linköping University, Computer Engineering, The Institute of Technology, 2014.
- [18] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI (2Nd Ed.): Portable Parallel Programming with the Message-passing Interface*. MIT Press, Cambridge, MA, USA, 1999.
- [19] Anders Nilsson. *Design of programmable multi-standard baseband processors*. PhD thesis, Linköping University, Computer Engineering, The Institute of Technology, 2007.
- [20] Andréas Karlsson. *Algorithm Adaptation and Optimization of a Novel DSP Vector Co-processor*, 2010.
- [21] Mike Johnson. *Superscalar microprocessor design*. prentice Hall, 1991.
- [22] Subbarao Palacharla, Norman P. Jouppi, and James E. Smith. Quantifying the Complexity of Superscalar Processors, 1996.
- [23] Texas Instruments. TMS320C64x technical overview. *Texas Instruments*, Feb, 2000.
- [24] Xtensa Microprocessor Tensilica. Overview Handbook.
- [25] Yiyang Tang, Yuke Wang, Jin-Gyun Chung, S. Song, and M. Lim. High-speed assembly FFT implementation with memory reference reduction on DSP processors. In *Electronics, Circuits and Systems, 2004. ICECS 2004. Proceedings of the 2004 11th IEEE International Conference on*, pages 547–550, 2004.
- [26] S. Ricci F. Guidi and P. Tortoli. Assembly Optimised FFT Algorithms for TMS320C62 Processors. In *Proc. Third European DSP Education and Research Conference*, September 2000.

- [27] Ujval J Kapasi, William J Dally, Scott Rixner, John D Owens, and Brucek Khailany. The Imagine stream processor. In *Computer Design: VLSI in Computers and Processors*, 2002. *Proceedings. 2002 IEEE International Conference on*, pages 282–288. IEEE, 2002.
- [28] Peter Raymond Mattson. *A Programming System for the Imagine Media Processor*. PhD thesis, Stanford University, Stanford, CA, USA, 2002. AAI3040045.
- [29] Johan Eker and Jörn W. Janneck. CAL Language Report: Specification of the CAL actor language, December 2003.
- [30] C. Kozyrakis, D. Judd, J. Gebis, S. Williams, D. Patterson, and K. Yelick. Hardware/compiler codevelopment for an embedded media processor. *Proceedings of the IEEE*, 89(11):1694–1709, Nov 2001.
- [31] David Judd, Katherine Yelick, Christoforos Kozyrakis, David Martin, and David Patterson. Exploiting On-chip Memory Bandwidth in the VIRAM Compiler. In *Intelligent Memory Systems*, volume 2107 of *Lecture Notes in Computer Science*, pages 122–134. Springer Berlin Heidelberg, 2001.
- [32] C. Kozyrakis and D. Patterson. Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks. In *Microarchitecture*, 2002. (MICRO-35). *Proceedings. 35th Annual IEEE/ACM International Symposium on*, pages 283–293, 2002.
- [33] Chenxin Zhang. *Dynamically Reconfigurable Architectures for Real-time Baseband Processing*. PhD thesis, Lund University, 2014.
- [34] M. Woh, Yuan Lin, Sangwon Seo, S. Mahlke, T. Mudge, C. Chakrabarti, R. Bruce, D. Kershaw, A. Reid, M. Wilder, and K. Flautner. From SODA to scotch: The evolution of a wireless baseband processor. In *Microarchitecture*, 2008. MICRO-41. *2008 41st IEEE/ACM International Symposium on*, pages 152–163, Nov 2008.

- [35] Jorge Nocedal and Stephen J. Wright. *Numerical optimization*. Springer series in operations research and financial engineering. Springer, New York, NY, 2. ed. edition, 2006.
- [36] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. Wiley-Interscience, New York, NY, USA, 1999.
- [37] Gurobi Optimization. Gurobi Optimizer Reference Manual. <http://www.gurobi.com/>, 2014.
- [38] Google. or-tools. <http://code.google.com/p/or-tools/>, 2014.
- [39] Krzysztof R. Apt and Mark Wallace. *Constraint Logic Programming Using Eclipse*. Cambridge University Press, New York, NY, USA, 2007.
- [40] JaCoP. <http://jacop.osolpro.com/>.
- [41] Gecode. <http://www.gecode.org/>.
- [42] E.S.H. Hou, N. Ansari, and Hong Ren. A genetic algorithm for multiprocessor scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 5(2):113–120, Feb 1994.
- [43] Hong Zhang, C. M. Tam, and Heng Li. Multimode Project Scheduling Based on Particle Swarm Optimization. *Computer-Aided Civil and Infrastructure Engineering*, 21(2):93–103, 2006.
- [44] Emad Elbeltagi, Tarek Hegazy, and Donald Grierson. Comparison among five evolutionary-based optimization algorithms. *Advanced Engineering Informatics*, 19(1):43 – 53, 2005.
- [45] Steven S. Muchnick. *Advanced compiler design & implementation*. Morgan Kaufmann, 1997.
- [46] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools*, volume 1009. Pearson/ Addison Wesley, 2007.

- [47] Andrew W Appel. *Modern compiler implementation in ML*. Cambridge university press, 1998.
- [48] Jörg Keller and Christoph W. Kessler. Optimized Pipelined Parallel Merge Sort on the Cell BE. In *Euro-Par 2008 Workshops - Parallel Processing*, volume 5415 of *Lecture Notes in Computer Science*, pages 131–140. Springer Berlin Heidelberg, 2009.
- [49] Rikard Hultén, Christoph W. Kessler, and Jörg Keller. Optimized On-Chip-Pipelined Mergesort on the Cell/B.E. In *Euro-Par 2010 - Parallel Processing*, volume 6272 of *Lecture Notes in Computer Science*, pages 187–198. Springer Berlin Heidelberg, 2010.
- [50] Christoph W. Kessler and Jörg Keller. Optimized Mapping of Pipelined Task Graphs on the Cell BE, January 2009.
- [51] D. Talla, L.K. John, and D. Burger. Bottlenecks in multimedia processing with SIMD style extensions and architectural enhancements. *Computers, IEEE Transactions on*, 52(8):1015 – 1031, aug. 2003.
- [52] P.T. Hulina, L.D. Coraor, L. Kurian, and E. John. Design and VLSI implementation of an address generation coprocessor. *Computers and Digital Techniques, IEE Proceedings -*, 142(2):145 –151, mar 1995.
- [53] Gang Ren, Peng Wu, and David Padua. Optimizing data permutations for SIMD devices. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, PLDI '06*, pages 118–131, New York, NY, USA, 2006. ACM.
- [54] Franz Franchetti and Markus Püschel. Generating SIMD vectorized permutations. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction, CC'08/ETAPS'08*, pages 116–131, Berlin, Heidelberg, 2008. Springer-Verlag.
- [55] Peter A. Milder, James C. Hoe, and Markus Püschel. Automatic Generation of Streaming Datapaths for Arbitrary Fixed Permuta-

- tions. In *Design, Automation and Test in Europe (DATE)*, pages 1118–1123, 2009.
- [56] Markus Püschel, Peter A. Milder, and James C. Hoe. Permuting streaming data using RAMs. *J. ACM*, 56:10:1–10:34, April 2009.
- [57] A. Tarable, S. Benedetto, and G. Montorsi. Mapping interleaving laws to parallel turbo and LDPC decoder architectures. *Information Theory, IEEE Transactions on*, 50(9):2002–2009, Sept 2004.
- [58] Dorit Naishlos, Marina Biberstein, Shay Ben David, and Ayal Zaks. Vectorizing for a SIMdD DSP architecture. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems, CASES '03*, pages 2–11, New York, NY, USA, 2003. ACM.
- [59] Libo Huang, Li Shen, and Zhiying Wang. Permutation optimization for SIMD devices. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 3849 –3852, 30 2010-june 2 2010.
- [60] J. H. Moreno, V. Zyuban, U. Shvadron, F. D. Neeser, J. H. Derby, M. S. Ware, K. Kailas, A. Zaks, A. Geva, S. Ben David, S. W. Asaad, T. W. Fox, D. Littrell, M. Biberstein, D. Naishlos, and H. Hunter. An innovative low-power high-performance programmable signal processor for digital communications. *IBM Journal of Research and Development*, 47(2.3):299–326, march 2003.
- [61] Libo Huang, Li Shen, Zhiying Wang, Wei Shi, Nong Xiao, and Sheng Ma. SIF: Overcoming the limitations of SIMD devices via implicit permutation. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, Jan 2010.
- [62] NVIDIA. *NVIDIA CUDA C Programming Guide*, July 2013.
- [63] NVIDIA. *CUDA C Best Practices*, July 2013.

- [64] Akira Nukada and Satoshi Matsuoka. Auto-tuning 3-D FFT library for CUDA GPUs. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09*, pages 30:1–30:10, New York, NY, USA, 2009. ACM.
- [65] Michael Gössel, Burghard Rebel, and Reiner Creutzburg. *Memory Architecture and Parallel Access*. Elsevier Science Inc., New York, NY, USA, 1994.
- [66] Joar Sohl, Jian Wang, Andréas Karlsson, and Dake Liu. Automatic Permutation for Arbitrary Static Access Patterns. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 215 –222, July 2012.
- [67] Andreas Björklund, Thore Husfeldt, and Mikko Koivisto. Set Partitioning via Inclusion-Exclusion. *SIAM J. Comput.*, 39:546–563, July 2009.
- [68] Joar Sohl, Jian Wang, Andréas Karlsson, and Dake Liu. Conflict-free data access for multi-bank memory architectures using padding. In *High Performance Computing (HiPC), 2013 IEEE Conference on*, pages 425–432, December 2013.
- [69] Andreas Krall and Sylvain Lelait. Compilation Techniques for Multimedia Processors. *International Journal of Parallel Programming*, 28(4):347–361, 2000.
- [70] Rainer Buchty, Vincent Heuveline, Wolfgang Karl, and Jan-Philipp Weiss. A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators. *Concurrency and Computation: Practice and Experience*, 24(7):663–675, 2012.
- [71] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Grösslinger, and Louis-Noël Pouchet. Polly-Polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, 2011.

- [72] K. Iwasawa and A. Mycroft. Choosing Method of the Most Effective Nested Loop Shearing for Parallelism. In *Parallel and Distributed Computing, Applications and Technologies*, 2007. PDCAT '07. Eighth International Conference on, pages 267–276, Dec 2007.
- [73] Manuel Hohenauer, Felix Engel, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. A SIMD optimization framework for retargetable compilers. *ACM Trans. Archit. Code Optim.*, 6:2:1–2:27, April 2009.
- [74] S. Maleki, Yaoqing Gao, M.J. Garzaran, T. Wong, and D.A. Padua. An Evaluation of Vectorizing Compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 372–382, Oct 2011.
- [75] M.A. Nichols, H.J. Siegel, and H.G. Dietz. Data management and control-flow aspects of an SIMD/SPMD parallel language/compiler. *Parallel and Distributed Systems, IEEE Transactions on*, 4(2):222–234, Feb 1993.
- [76] Timothy A. Budd. An APL Compiler for a Vector Processor. *ACM Trans. Program. Lang. Syst.*, 6(3):297–313, July 1984.
- [77] F. Franchetti and M. Puschel. A SIMD vectorizing compiler for digital signal processing algorithms. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 7 pp–, April 2002.
- [78] Mehmet Ali Arslan and Krzysztof Kuchcinski. Instruction selection and scheduling for DSP kernels. *Microprocessors and Microsystems*, 38(8, Part A):803 – 813, 2014. 2013 edition of the Euromicro Conference on Digital System Design (DSD 2013).
- [79] OpenCL. <http://www.opencl.com/>.
- [80] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

- [81] Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra, and Steven Huss Lederman. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995.
- [82] Murray Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991.
- [83] Markus Ålind, Mattias V. Eriksson, and Christoph W. Kessler. BlockLib: a skeleton library for cell broadband engine. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 7–14, New York, NY, USA, 2008. ACM.
- [84] F# programming language. <http://www.fsharp.org/>.
- [85] Haskell programming language. <http://www.haskell.org/>.
- [86] Objective Caml. <http://caml.inria.fr/ocaml/index.en.html>.
- [87] Pascal Brisset and Nicolas Barnier. FaCiLe: a Functional Constraint Library, 2001.
- [88] ITU-T. *ITU-T Rec. H.262 (2000 E)*, February 2000.
- [89] Yanjun Zhang, Wenbiao Zhou, Zhenyu Liu, Siye Wang, and Dake Liu. A High Definition Motion JPEG Encoder Based on Epuma Platform. *Procedia Engineering*, 29(0):2371 – 2375, 2012. 2012 International Workshop on Information and Electronics Engineering.