Giulio Stramondo

MEMORY SYSTEM DESIGN FOR APPLICATION-SPECIFIC HARDWARE

Giulio Stramondo

MEMORY SYSTEM DESIGN FOR APPLICATION-SPECIFIC HARDWARE

UNIVERSITY OF AMSTERDAM

MEMORY SYSTEM DESIGN FOR APPLICATION-SPECIFIC HARDWARE

GIULIO STRAMONDO

NOKIA Bell Labs

This work has received funding from the EU Horizon 2020 research and innovation programme under grant No 671653.

The work in Chapters 5 and 6 was partially done during an internship at Nokia Bell Labs.

Copyright © 2020 by Giulio Stramondo.

Thesis template: `classicthesis` by André Miede and Ivo Pletikosić.
Printed and bound by Ipskamp Printing.

MEMORY SYSTEM DESIGN FOR APPLICATION-SPECIFIC HARDWARE

Academisch Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. ir. K.I.J. Maex
ten overstaan van een door het College voor Promoties
ingestelde commissie,
in het openbaar te verdedigen in de Agnietenkapel
op donderdag 8 april 2021, te 16.00 uur

door

Giulio Stramondo

geboren te Catania, Italy

Promotiecommisie

Promotor:	prof. dr. ir. Cees T.A.M. de Laat	Universiteit van Amsterdam
Supervisor:	dr. ir. Ana Lucia Varbanescu	Universiteit van Amsterdam
Overige leden:	prof. dr. Dirk Stroobandt	Universiteit Gent
	prof. dr. Henk Sips	Technische Universiteit Delft
	prof. dr. Alfons Hoekstra	Universiteit van Amsterdam
	prof. dr. Pieter W. Adriaans	Universiteit van Amsterdam
	prof. dr. Andy Pimentel	Universiteit van Amsterdam
	dr. Marco D. Santambrogio	Politecnico di Milano
	dr. Francesco Regazzoni	Universiteit van Amsterdam

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth [39]

ACKNOWLEDGMENTS

This PhD thesis would not exist if it was not for the help of some people in my life. It might have been thanks your friendship, your mentoring, or simply your support that I went through these past 4 years. In this section I will try to thank name all of the people who helped me, although one of the issues of any finite list is that it is prone to error. If I forget to mention you, please let me know and I will gladly offer you beers until you forget about it!

The first person that I would like to thank is Ana, my supervisor. I would probably need a separate book to properly thank her for everything, but let me try to compress it in a paragraph; thank you for always being able to find time for me and for helping me to grow as a scientist, and most importantly as a person. Thank you for transmitting me your enthusiasm about the research process. Your optimism helped me see the bright side of things (seeing the glass half full). I admire your passion about teaching, and I cherish the laughs, the chocolate, the nerdy jokes and the occasional glass of wine we shared (which instead was rather half empty).

Thank you Cees, for accepting me as your PhD student and for your time and inputs during the process. I am still amazed by the amount of time you are able to dedicate to each and every one of your students.
Thank you Catalin, for your supervision during my first two years of PhD, for your support and advice.
Thanks to Marco Santambrogio, who was my master thesis supervisor at the Politecnico di Milano as well as the one who

pushed me to start a PhD. I also had the pleasure to work with him during my PhD as part of the EXTRA project, and towards the end as part of my PhD committee. I did learn a lot from you, and I would not be writing this thesis if it was not for you, thank you!

I would like to thank all the members of the EXTRA Project, for all the exciting conversations and fruitful collaborations we had. Special thanks for our close interaction during the project to Marco Rabozzi, Lorenzo Di Tucci, Luca Stornaiuolo and Giuseppe Natale, from the Politecnico di Milano, Amit Kulkarni, from the Gent University, Stefano Cardamone, from the University of Cambridge, and Andreas Brokalakis, from Synelixis.
I would also like to thank the Nokia Bell Labs of Antwerp for the great opportunity of an internship, with special thanks to Bartek Kozicki and Manil Dev Gomony. Manil, thank you for your help, guidance, the exciting scientific conversations and your presence during and after my internship. Thank you to my PhD committee for your time and your useful feedback, Prof. Dirk Stroobandt, Prof. Henk Sips, Prof. Alfons Hoekstra, Prof. Pieter W. Adriaans, Prof. Andy Pimentel, Dr. Marco D. Santambrogio and Dr. Francesco Regazzoni.

Moving to a new country and all the complexity related to successfully going through the ups and downs of a PhD is without doubts a thrilling and tough experience. Thanks to the guys at the SNE Lab I quickly felt welcome into this new universe. I would like to thank you all, however I feel that I am destined to forget someone. If I did, please let me know and allow me to apologize. Thank you, Adam, Ameneh, Ana (Oprescu - the other Ana), Andy, Benny, Clemens, Dolly, Francesco, Giovanni, Jamila, Marijke, Mark, Merijn, Milen, Mostafa, Paola, Pieter, Peter, Simon, Tim, Yuri and Zimming.

During the last 4 years, I got to meet many amazing fellow researchers/colleagues, of whom many were going/went/are going through the same share of extreme emotions to complete a PHD. To my chinese friends and colleagues: Jun, thanks for allowing me to be at your side in your important days, introducing me to your wonderful wife and your baby girl, and all the wonderful hot pots we shared; Huan and Yang, thank you for sharing with

me coffees, laughs and pijiu. To all of you, my most sincere and warm Ganbei! To Uraz, thank you for being always the last drop! Thank you Ben (and Cecile) for your help and friendship. Ben... please never offer to my son a huge inflatable unicorn. To all the "unusual suspects" thank you for making the winters in the Netherlands bearable and the summer endless barbecues! Thank you Regi and Daniela for the picnics at the park, the fun times on the slopes, at the beach, the dinners, or the game nights, for the almost Maltese holidays, for being so supportive and being so close to me and Eugenie. Thank you Ralph and Rian, for the Sinterklaas nights, the good advices and help with the thesis, for introducing me to the Dutch culture and the bouldering days. To Spiros and Angeliki, for the snow trips, windsurf talks and housewarming parties. To Joe, for always being my best man, teaching me how to wing it, for the time we shared on the snow, sailing and playing. To Mary, special thanks for the burritos, the cat allergies, amazingly full glasses of wine we shared (I am sorry your glasses will be half-full from now on), and for always sharing couches when in times of need. To Julius and Helena, for training with me during the fire drills (3 minutes...) and even sharing your bed with me (yes, sorry Helena). To Misha and Tammy, for the great conversations and dinners, the chess games, the boxing and the broken necks (boxing and broken necks are weirdly unrelated). To Lukasz, thank you for all the "Mehhh", the run in the forest at -10 degrees, the improvised sailings, and the wasted days at the gym. To Xin, thank you for our talks, runs and drawings tricks you taught me. Each of you made these four years in the Netherlands golden and I will cherish these memories. Your pictures will always be on my fridge (and I will probably need to buy a bigger fridge first).

I would also like to thank my friends back home for their constant support during these years. Even if I was physically far, they never made me feel distant. Thank you, Giulio (Milone), Rosita, Carlo, Stefania, Roberto, Martina, Paolo, Giorgia, Stefano, Marta, Jacopo, Eliana and Daniele.

Special thanks to my family, that supported me, encouraged me, cheered me up or waited for me to stop working late. Thank you Eric, Line, Margaux, Gabi, Roberto, Francesca, Sergio, Maria

Luisa, Lorenzo, Giovanni, Nicolo', Silvia. Finally, thank you Mom for your (maybe sometimes a bit excessive) love, your care and fair share of optimism. Thank you Dad for teaching me to always try to push myself further into uncharted territories and for your support. And thank you Eugenie, for being by my side, in the ups and downs, always being able to let me pick myself up with your loving smile.

CONTENTS

1 INTRODUCTION 1
1.1 Research questions and approach 3
1.2 Structure of this thesis 5
2 BACKGROUND AND TERMINOLOGY 7
2.1 Memory and processing systems 7
2.1.1 Parallel Memories 8
2.1.2 Types of Parallel Memories 9
2.1.3 The target application 10
2.2 Memory Technologies 12
2.3 Hardware Platforms 15
2.3.1 Field Programmable Gate Arrays (FPGAs) 15
2.3.2 Spatial Processor 17
3 A PARALLEL SOFTWARE CACHE 19
3.1 Introduction 19
3.1.1 The Polymorphic Register File 22
3.2 Design and Implementation 24
3.2.1 End-to-end design 24
3.2.2 From PolyMem to MAX-PolyMem 25
3.2.3 Productivity Analysis 27
3.3 Design Space Exploration and Results 28
3.3.1 Design Space Exploration setup 28
3.3.2 Memory Performance 29
3.3.3 Resource utilization 31
3.4 STREAM-Copy: Bandwidth Benchmarking 34
3.5 Related Work 37
3.6 Summary 38
4 APPLICATION-CENTRIC PARALLEL MEMORIES 41
4.1 Introduction 42
4.2 Preliminaries and Terminology 43
4.2.1 Parallel Memories 44
4.2.2 The Application 44
4.3 Scheduling an Application Access Trace to a PM 47
4.3.1 The set covering problem 47
4.3.2 From Concurrent Accesses to Set Covering 47

4.3.3 An Heuristic Approach 49
4.3.4 The Complete Approach 50
4.4 Evaluation 51
4.4.1 Experiment Setup 51
4.4.2 Results 52
4.5 Experiments and Results 55
4.5.1 MAX-PolyMem 55
4.5.2 Sparse STREAM 56
4.5.3 Results 57
4.6 Related Work 58
4.7 Summary 59
5 COMPUTE AND MEMORY SYSTEM CODESIGN 61
5.1 Introduction 62
5.2 The μ-Genieframework 64
5.2.1 Model of Execution 64
5.2.2 The L2 Memory Model 65
5.3 μ-Genie: Inputs 66
5.3.1 Application 66
5.3.2 Configuration Parameters 66
5.4 μ-Genie: Analysis 67
5.4.1 L2 Memory Read and Write Modeling 67
5.4.2 Data Dependency Analysis 69
5.4.3 PE allocation with Modified Interval Partitioning 71
5.4.4 Most Parallel and Most Sequential Architectures 72
5.5 μ-Genie: Design Space Exploration (DSE) 73
5.5.1 Architecture Tradeoffs 74
5.6 Architectural Template 74
5.7 Summary 76
6 DSE FOR CODESIGNED COMPUTE AND MEMORY SYSTEMS 77
6.1 Multi-Configuration Design Space Exploration 77
6.2 Case Studies 78
6.2.1 Single configuration DSE 78
6.2.2 MRAM vs SRAM Level 2 Memory 79
6.2.3 Different Matrix Dimensions 80
6.3 Related Work 80
6.4 Summary 82

7 conclusion 87
7.1 Main Findings 87
7.2 Main contributions 90
7.3 Future Research Directions 90

bibliography 93
publications 103
software and data 105
summary 107
samenvatting 111

LIST OF FIGURES

Figure 1.1 McCalpin's CPU Speed vs. Bandwidth [49] 2
Figure 1.2 Structure of the thesis 5
Figure 2.1 Memory system and Processing systems 8
Figure 2.2 DRAM memory cell 12
Figure 2.3 SRAM memory cell 13
Figure 2.4 Magnetoresistive RAM memory cell 14
Figure 2.5 Spatial Architectures classification [59]. 18
Figure 3.1 Envisioned system organization using PolyMemas a parallel cache. 20
Figure 3.2 PolyMem supported access patterns. 23
Figure 3.3 The Block-diagram of our MAX-PolyMem Implementation. 25
Figure 3.4 Write bandwidth. 31
Figure 3.5 Read bandwidth (aggregated). 31
Figure 3.6 Logic utilization. 32
Figure 3.7 LUT Utilization. 32
Figure 3.8 BRAM Utilization. 33
Figure 3.9 The implementation of the STREAM benchmark for MAX-PolyMem. All transfers between host (the CPU) and PolyMem (on the FPGA) are done via the PCIe link. 35
Figure 3.10 Copy bandwidth (aggregated). 37
Figure 4.1 Customizing parallel memories. Our research focuses on the mapping of the access trace from the application to the parallel access patterns of the parallel memory. 43
Figure 4.2 An overview of our complete approach. 51
Figure 4.3 Evaluation of the ILP and Heuristic (HEU) results. 54
Figure 4.4 The implementation of the STREAM benchmark for MAX-PolyMem (figure updated from [14]). All transfers between host (the CPU) and PolyMem (on the FPGA) are done via the PCIe link. 56

Figure 4.5 The performance results (measured, predicted, and ideal) for the 10 different variants of the STREAM benchmark. The horizontal lines indicate the theoretical bandwidth of MAX-PolyMem, configured with 8-byte data, 8 lanes, and 2 (for Copy and Scale) or 3 (for Sum or Triad) parallel operations. Running at 100MHz, MAX-PolyMem can reach up to 12.8GB/s for 1-operand benchmarks and up to 19.6GB/s for 2-operand benchmarks. 59
Figure 5.1 Difference between state-of-the-art design flow typically used for traditional application-specific processors and the proposed μ-Genie design flow for spatial processors. 63
Figure 5.2 μ-Genie Framework. 64
Figure 5.3 The system under analysis. 64
Figure 5.4 A Data Dependency Graph: inverse triangles represent input data, obtained from the *load* instructions; ovals describe operations on data; the triangle at the bottom represents the result, derived from a *store* instruction. Highlighted, a chain of associative operations before being optimized by the *DDA* module (5.4.2). 70
Figure 5.5 Example of architectures generated from a matrix vector multiply application of size 5x5. The MostPar (a), an intermediate architecture (b) and the MostSeq (c). 74
Figure 5.6 Functional Unit template. PE1-PE4 represent "parent" PEs that generate input data. IM is an internal Instruction Memory,where the PE stores the operations to be performed. RFs are internal Register Files, which store reuse data and inputs to be used in the future. OP is the hardware unit actually performing the PE operation. 75

Figure 6.1 Each point represents one μ-Genie spatial processor. Different shapes (in 6.1b and 6.1c) identify different input configurations. 6.1a shows the architecture's Energy over Latency in clock cycles generated from a single configuration of a matrix vector multiplication of size 5×5. Note that (a) presents all designs, while (b) and (c) only include the Pareto-optimal designs. 84
Figure 6.2 Energy Pareto optimal architectures generated by μ-Genie for different sizes of Matrix Vector multiplication 5x5 - with latency ranging from 0 to 1000,10x10 - having latency between 1000ns and 2500ns , and 15x15- with latency above 2500ns. Each point corresponds to an architecture generated by the framework. 85

LIST OF TABLES

Table 2.1 Comparison between 28nm SRAM and MRAM memories 14
Table 3.1 The PRF memory access schemes 22
Table 3.2 Productivity analysis. 26
Table 3.3 PolyMem Design Space Exploration Parameters. 29
Table 3.4 MAX-PolyMem Maximum Clock Frequencies [MHz]. 30
Table 4.1 The 10 variants of the STREAM benchmark and the predicted performance of the calculated schedules for two schemes (ReRo and RoCo). The other schemes are omitted because they are not competitive for these patterns. In the patterns, only the R elements need to be read. 57

Table 5.1 Definition of symbols used in the equations 68
Table 6.1 Comparison with related work. 82

LISTINGS

Listing 5.1 Modified Interval Partitioning (MIP) Algorithm 72
Listing 5.2 Design Space Exploration 73

ACRONYMS

AI Artificial Intelligence

AGU Address Generation Unit

ALAP AS Late As Possible

ALU Arithmetic Logic Unit

ASAP As Soon As Possible

BRAM Block Random Access Memory

CPU Central Processing Unit

DDG Data Dependency Graph

DRAM Dynamic Random Access Memory

eDRAM Embedded Dynamic Random Access Memory

DMA Direct Memory Access

DFE Data Flow Engine

DSE Design Space Exploration

FeRAM Ferroelectric Random Access Memory

FLOPs Floating Point Operation Per second

FPGA Field-Programmable Gate Array

GPU Graphics Processing Unit

HDL Hardware Description Language

HLS High Level Synthesis

HPC High Performance Computing

IDE Integrated Development Environment

ILP Integer Linear Programming

IM Instruction Memory

L1M Level 1 Memory

L2M Level 2 Memory

LOC Lines Of Code

LUT Look Up Table

MAF Module Assignment Function

MIP Modified Interval Partitioning

MM Matrix Matrix Multiplication

MMF Memory Mapping Function

MRAM Magnetoresistive Random Access Memory

MTJ Magnetic Tunnel Junction

MV Matrix Vector Multiplication

NP Nondeterministic Polynomial time

PE Processing Element

PCM Phase Change Memory

PCI Peripheral Component Interconnect

PM Parallel Memory

PRF Polymorphic Register File

RAM Random Access Memory

RRAM Resistive Random Access Memory

RF Register File

RTL Register Transfer Level

RQ Research Question

SIMD Single Instruction Multiple Data

SRAM Static Random Access Memory

VHDL Very High Speed Integrated Circuit Hardware Description Language

1 INTRODUCTION

Computers are a driving factor of modern scientific research. The computing capability we have nowadays on a single laptop are higher than the one of the most powerful supercomputer we had two decades ago. The rapid development in the silicon industry allowed all the fields of research to increase their output, and perform studies which would be unfeasible or too costly otherwise. To mention a few examples, we witnessed a dramatic improvement in the fields of genetics - decoding almost completely the human DNA [20], and in neuroscience - where the task of modeling the organization and behaviour of the human brain seems now within reach [46]. The Artificial Intelligence (AI) explosion which we are witnessing today can also be, at least in part, attributed to the current advances in computing power.

However, the current pace at which computing technologies improves is likely to slow down [25]. For many years, techniques such as frequency scaling, and miniaturization of the transistors have been leveraged successfully to gain performance. But these techniques are now close to their physics limits, forcing computer scientists and engineers alike to find alternative ways to improve the performance of computing systems. Parallel computing, distributed computing, and the use of heterogeneous platforms are becoming predominant ways to increase computing power [67].

Broadly speaking, the architectures of these modern computing systems can still be seen, at the highest level of abstraction, as combining a memory and a processing system. Historically, research on processing systems has been predominant. However, modern architectures are often limited by their memory system. In fact, on-chip memory systems occupy over 30% of the chip-area [66] and consume a significant fraction (30% to 60%) of the overall energy dissipated [82]. Moreover, for many applications, the memory system represents a fundamental performance bottleneck [54]. This bottleneck appears because the performance of the processing system increases much faster than the per-

formance of the memory system, creating over time a growing performance gap. This gap has been captured as the "memory wall" by W.A. Wulf [85]. Figure 1.1 shows performance trends of CPUs and memory over time: we observe that the MFLOPs of a CPU increase, on average, with 50% every year, while the memory bandwidth only increases, on average, with 35% [49]. This means, in practice, that applications spend increasingly more time waiting for data then actually using it for computing.

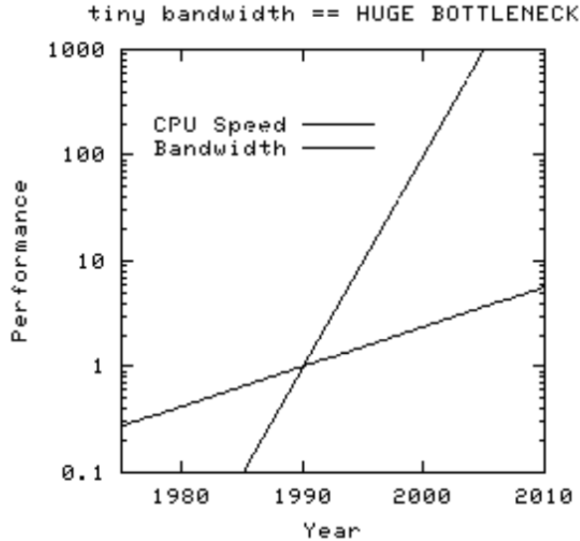


Figure 1.1: McCalpin's CPU Speed vs. Bandwidth [49]

The discovery of the *memory wall* boosted memory systems research. For example, researchers proposed in-memory computation [2, 89], novel memory technologies (e.g. magnetoresistive RAM [55], ferroelectric RAM [11], resistive RAM [68]), and multi-banked (or parallel) memories [12, 45].

Our work also proposes solutions to alleviate the memory wall problem. Specifically, we study the *design and implementation of parallel memory systems optimized for target applications*. The core idea behind the use of parallel memories is that, by increasing the number of banks in the memory system, and accessing them concurrently, it is possible to significantly increase the bandwidth of the memory system. However, the data layout in the memory banks, combined with the access pattern used by the target ap-

plication(s) play a crucial role in the effective attainable memory system performance.

1.1 RESEARCH QUESTIONS AND APPROACH

The main Research Question (RQ) addressed in this work is:

RQ: How can we design and implement efficient application-specific parallel memories?

To answer our main RQ, we pursue the following four detailed research questions:

- **RQ1**: What is a feasible design for a configurable hardware parallel memory?

  Field Programmable Gate Arrays (FPGAs) seem to be ideal platforms to prototype parallel memories, because they have on-chip RAM blocks that can be combined into a single memory space, while still being accessible concurrently, like independent memory banks. To answer **RQ1**, we analyse the feasibility of designing a configurable parallel memory on such architectures. We further provide a configurable *template* of a parallel memory that can be adjusted to provide high-bandwidth memory accesses for a given application. Finally, we present a prototype of this configurable hardware memory, deployed on a Maxeler-based FPGA platform, and demonstrate its peak performance capabilities.

- **RQ2**: Can we define and implement a systematic, application-centric method to configure a hardware parallel memory?

  To configure a parallel memory such that it improves the performance of a target application, we must analyse the in-memory data layout and the memory access pattern of the application. Within **RQ2**, we identify and address the challenges of configuring a parallel memory given an application. Building on top of the proposed FPGA-based template (RQ1), we provide a systematic approach to determine the most efficient configuration for a given application access pattern. We supplement this methodology with an analytical performance model, which predicts the speed-up

and efficiency gain of using a parallel memory for a given application. Finally, we demonstrate how this approach leads to application-specific parallel memories, customized to maximum possible efficiency, for applications with both dense and sparse memory accesses.

- **RQ3**: Is there a systematic way to codesign efficient processing and memory systems from a given application?

  The semantics of an application depends on the correct execution of dependent instructions. It is thus possible to modify the access pattern of a given application by reordering its execution, while still maintaining the data dependencies. Changes in the application execution are then reflected in the application access pattern. In the scope of **RQ3**, we focus on codesigning the parallel memory and the application, in an effort to match the parallel-memory data-organization with the access pattern of an application. We propose a method for this codesign process, and implement a complete framework to support this method. We finally demonstrate how the codesigned architectures can be implemented in practice, and we assess their performance.

- **RQ4**: Is design exploration a feasible method to codesign parallel-memory computing systems?

  The codesign of the memory and processing systems can be seen as a multi-objective optimization problem over a space of possible designs. Within **RQ4**, we focus on the definition of the dimensions of the design space and on its systematic exploration. We are the first to include, next to traditional dimensions like area, energy and latency, also memory technology. We further prototype and demonstrate a systematic design-space exploration methodology to expose novel tradeoffs, including these new dimensions. Using this DSE, we are able to provide in-depth quantitative analysis of tradeoffs such area-energy or energy-memory technology.

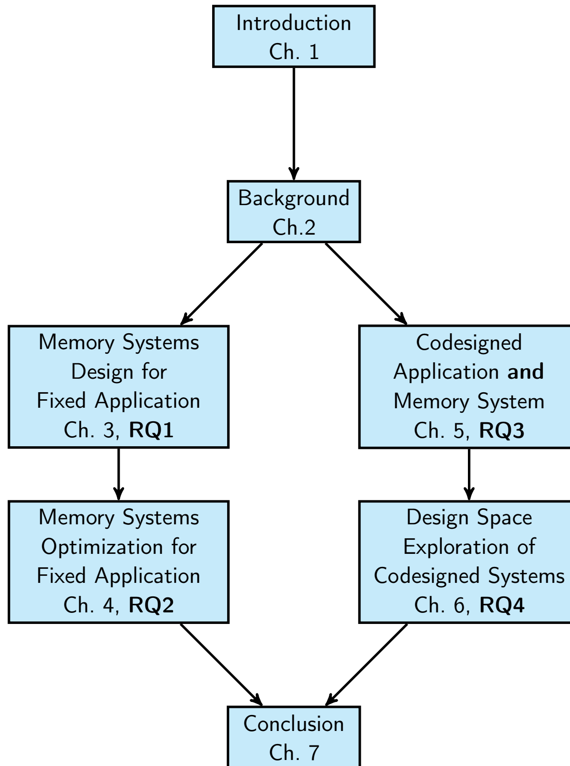


Figure 1.2: Structure of the thesis

1.2 STRUCTURE OF THIS THESIS

This thesis evaluates two methodologies for designing application-specific parallel memory systems (Figure 1.2). In the first methodology, we propose a configurable design of a parallel memory system - PolyMem (Chapter 3), which is then optimized *for a fixed* target application access pattern (Chapter 4).

The second methodology evaluates the feasibility of *codesigning* the memory *and* the computing systems together. In this approach, we optimize, in the same time, the data placement in the parallel memory *and* the application access pattern (as opposed to, *for* in the previous approach) - Chapter 5. Chapter 6 demonstrates the feasibility and flexibility of *codesigned* memory and computing systems through extensive design space exploration and analysis of the features such codesigned systems can offer.

2 BACKGROUND AND TERMINOLOGY

In this chapter we present the essential background concepts, terminology, and basic definitions necessary to understand the remainder of this work.

2.1 MEMORY AND PROCESSING SYSTEMS

A computing system can be modeled as composed by a memory system and a processing system, as shown in Figure 2.1. The performance of the memory (processing) system can be expressed according to the amount of data produced (consumed) per unit of time. In an ideal scenario both systems produce and consume data at the peak of their capability, and the production and consumption rates match.

However, in most cases, in practice, the production and consumption rates between the two systems do not match. In such cases, the overall performance of the computing system is determined by the system having the lowest rate - i.e., the *bottleneck*. When the bottleneck is the processing (consumption) rate, the system is said to be *compute bound*. On the other hand, the system is *memory bound* if the memory system produces data at a lower rate than the computing system is able to consume. Conceptually, a system's performance can be improved by removing the bottleneck: optimize the cores (throughput, latency) if the processing capacity system is limiting the performance, or optimize the memory (bandwidth) if the memory system is lagging behind. Therefore, a crucial step when aiming to improve the performance of a computing system is to understand where the bottleneck is.

While we can reason about the theoretical production and consumption rates of the two sub-systems (processing and memory), when these are used to execute a real application, the behaviour of the application will affect where the bottleneck appears. Specifically, a *data-intensive* application would stress more the capability

of the memory system, leading to a "memory-bound" execution; similarly, running a *compute intensive* application on the same system will likely use the processing system much more, and it will likely lead to a "compute-bound" behaviour. Therefore, the way an application behaves plays a crucial role in the overall performance of the computing system.

The concepts expressed above are combined in the **roofline model**[84], which provides a quantitative way to analyse the (computing system, application) ensemble: it can identify the bottleneck of the ensemble by using the properties of *both* the application and the computing system, and can provide numerical upper bounds for the peak performance that can be achieved by this ensemble.

However, the roofline model provides no optimization solution. Instead, it is the user's responsibility to address the bottleneck: either improve the application implementation or the underlying compute system. In this thesis, we focus on the latter, and discuss ways in which the memory system can be fundamentally improved by concurrently using multiple memory banks (as shown in Figure 2.1). Consequently, it is likely that our solutions are beneficial for memory-bound applications and systems.

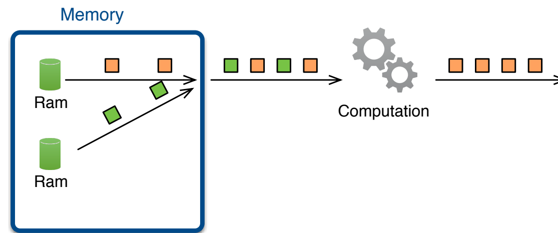


Figure 2.1: Memory system and Processing systems

2.1.1 *Parallel Memories*

Definition 1 (Parallel Memory) *A Parallel Memory (PM) is a memory that enables access to multiple data elements in parallel.*

A parallel memory can be realized by combining a set of independent memories - referred to as *sequential memories*, but also as *lanes* or *banks*. The *width of the parallel memory*, identified by

the number of sequential memories used in the implementation, represents the maximum number of elements that can be read in parallel. The *capacity* of the parallel memory refers to the amount of data it can store.

2.1.2 *Types of Parallel Memories*

Intuitively, a parallel memory (PM) is a memory that enables read and write operations for multiple data elements in parallel. The implementation of a parallel memory always relies on using M sequential memory blocks. However, depending on how the information is stored and/or retrieved from memory, we distinguish three types of parallel memories: redundant, non-redundant, and hybrid.

- **Redundant PMs** The simplest implementation of a PM is a fully redundant one, where all M sequential memory blocks contain fully replicated information. The benefit of such a memory is that it allows an application to access any combination of M data elements in parallel. However, such a solution has two major drawbacks: first, the total capacity of a redundant PM is M times lower than the combined capacities of all its sequential memories and, second, parallel writes are very expensive: whenever a data item needs to be updated, it needs to be updated in all memories to guarantee information consistency.
  To use such a memory, the application requires minimal changes, and the architecture is relatively simple to manage.

- **Non-Redundant PMs** Non-redundant parallel memories completely avoid data duplication: each data item is stored in only one of the M sequential memories. The one-to-one mapping between the coordinate of an element in the application space and a memory location is part of the memory configuration. These memories can use the full capacity of all the memory resources available, and data consistency is guaranteed by avoiding data replication, making parallel writes feasible as well. The main drawbacks of non-redundant parallel memories are that they require specific hardware to perform the mapping, and they restrict

the possible parallel accesses: if two elements are stored in the same sequential memory, they cannot be accessed in parallel (see Section 4.2.2).

There are two major approaches used to implement non-redundant PM: (1) use a set of predefined mapping functions that enable parallel accesses in a set of predefined shapes [16, 29, 30, 42], or, (2) derive an application-specific mapping function [83, 88]. For the first approach, the application requires additional analysis and potential changes, while the architecture is relatively fixed. For the second approach, however, a new memory architecture needs to be implemented for every application, potentially a more challenging task when the parallel memory is to be implemented in hardware.

- **Hybrid PMs** Besides the two extremes discussed above, there are also hybrid implementations of parallel memories, which combine the advantages of the two previous approaches by using partial data redundancy [31]. Of course, in this case, the challenge is to determine which data should be replicated and where. In turn, this solution requires both application and architecture customization.

Our work focuses on *non-redundant* parallel memories. Non-redundant parallel memories can use the full capacity of all the memory resources available, while data consistency is guaranteed by avoiding data replication. However, these parallel memories restrict the possible parallel accesses: only elements stored in different memories can be accessed in parallel (see Section 4.2.2).

2.1.3 *The target application*

Across this thesis, we use the term *application* to refer to the entity using the PM to read/write data. The application can be implemented in a hardware element directly connected to the PM, or as a software application, interfaced with the PM.

Without loss of generality, we will consider the data of an application to be stored in an array A of N dimensions. Each data element can then be identified by a tuple containing N coor-

dinates $I = (i_0, i_1, ..., i_{N-1})$, which are said to be the coordinates of element $A[I] = A[i_0][i_1]...[i_{N-1}]$ in the *application space*.

An application *memory access* is a read/write operation which accesses $A[I]$. A *concurrent access* is a set of memory accesses, $A[I_j], j = 1..P$, which the application can perform concurrently. An *application memory access trace* is a temporal series of *concurrent accesses*. Finally, a *parallel memory access* is an access to multiple data elements which actually happens in parallel.

Information such as the memory access trace of an application needs to be extracted through applications analysis. There are two different ways to analyze an application: *static* and *dynamic* analysis. Static analysis extracts information from the application source code. For example, the amount of iterations within a loop can be derived from the initialization of the loop iterator, termination condition, and the loop iterator increment. Instead, dynamic analysis involves the execution of the application to record the data of interest. It is useful to use dynamic analysis in the context of data-dependent applications, where the control-flow mutates according to input data, and therefore it is impractical to perform static analysis. To analyse an application dynamically, the code is *instrumented*, i.e., specific functions are run during the execution to log the application behaviour. In this work, we focus specifically on "static applications" - i.e., applications that can be characterized by static analysis.

Definition 2 (Parallel Access Conflict) *A parallel access has a conflict if at least two of the accessed data elements are stored in the same memory bank. A conflict prevents a set of memory accesses to be carried out in a completely concurrent manner.*

Definition 3 (Conflict-Free Parallel Access) *A set of Q memory accesses, $A[I_0]..A[I_{Q-1}]$, constitutes a conflict-free parallel access if:*

$$\forall (A[I_i], A[I_j])$$

$$\textit{where } i \neq j, 0 \leq i, j \leq Q-1, Q = M$$

$$loc(A[I_i]) = (m_i, addr_i), loc(A[I_j]) = (m_j, addr_j)$$

$$m_i \neq m_j.$$

Intuitively, given an application with a certain access pattern, depending on how data are stored in the banks of a parallel memory, a concurrent access of an application could request elements stored in the same bank. If this happens, the concurrent access is a conflicting access, and therefore needs to be split in multiple accesses, which leads to performance degradation.

Chapter 4 expands on the concept of conflicts, and proposes a methodology to generate data layouts in a parallel memory that minimize conflicts for a given application.

2.2 MEMORY TECHNOLOGIES

This section will give a brief introduction of the most common technologies currently used to build memories.

Memory technologies can be divided in two main classes: *volatile* and *non-volatile*. A volatile memory needs power to retain information, i.e., when the memory is not powered the data stored is lost.

Volatile memories

Two examples of volatile memories are the Dynamic Random Access Memory (**DRAM**) and the Static Random Access Memory (**SRAM**). Both these memories are based on *cells*.

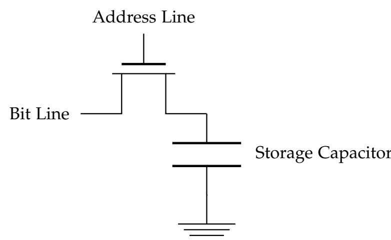


Figure 2.2: DRAM memory cell

A DRAM cell, able to store one bit of information, is realized using 1 transistor and 1 capacitor (shown in Figure 2.2); the capacitor is used to store the actual information, while the transistor can be used to charge and discharge the capacitor. The address line is used to select the DRAM cell: once the cell is selected, the charge stored in the capacitor is sent to the Bit line. The capacitor

produces a *leakage current*, that over time causes the loss of the charge stored. Thus, the charges contained in the capacitor need to be periodically refreshed.

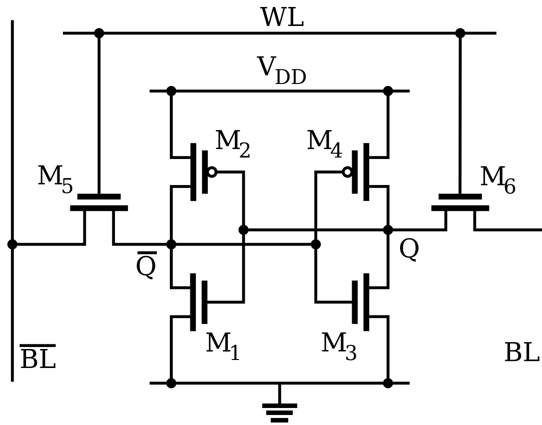


Figure 2.3: SRAM memory cell

An SRAM cell is realized (normally) using 6 transistors (Figure 2.3). Of these, 4 transistors form cross-coupled inverters having two stable states, which are used to represent information, and the additional 2 transistors provide access to the cell. The information in the SRAM cell is maintained as long as the cell is powered, removing the need to periodically refresh the stored data. The removal of the data refresh is one reason for which SRAM memories tend to be faster than DRAM memories. However, the use of more transistors implies that SRAM cells can store less data than DRAM cells, given the same amount of area.

Non-volatile memories

Non-volatile memories are capable to store data even when they are powered off. This implies that when there are no accesses to the memory, it does not need to consume power, hence, they are usually more power efficient than volatile memories. There are different technologies used to implement non-volatile memories (e.g. Magnetoresistive RAM (MRAM) [55], Ferroelectric RAM (FeRAM) [11], resistive RAM [68]). Figure 2.4 shows the structure of a MRAM cell [27], consisting of one transistor and a Magnetic Tunnel Junction (MTJ). The MTJ is used to store the information,

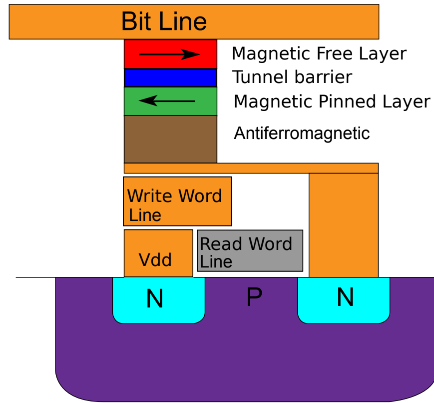


Figure 2.4: Magnetoresistive RAM memory cell

	1Mb 28nm MRAM[1]	1Mb 28nm SRAM[2]
Read Energy (pJ/bit)	0.68	2.57
Write Energy (pJ/bit)	4.5	2.67
Read Latency (ns)	2.8	1.022
Write Latency (ns)	20	1.022
Cell area (mm^2)	0.214	0.220
Leakage Energy (pJ/clock)	~0	1.98

[2] Q. Dong et al. ISSCC 2018, [3] TSMC

Table 2.1: Comparison between 28nm SRAM and MRAM memories

and consists of a Magnetic Free Layer, a Tunnel Barrier and a Magnetic Pinned Layer. The magnetic pinned layer has a fixed magnetic polarization, while the polarization of the magnetic free layer can be changed applying an external magnetic field. If the magnetization of the free and pinned layer are parallel to each other, the electrons will easily be able to tunnel through the Tunnel Barrier, otherwise they will encounter a strong resistance. Thus, the information stored in the MTJ can be read by measuring its resistance, and written by correctly magnetizing the magnetic free layer.

A characteristic of non-volatile memory technology is the latency and power imbalance between read and write operations. Table 2.1 shows a comparison between 28nm MRAMs and SRAMs, and illustrates that while read operations on MRAMs are comparable to SRAMs, write operations have higher latencies and energy consumption.

Chapters 5 and 6 define a methodology to take into account the read and write imbalance when designing memory systems, combining the use of SRAM and MRAM as use case.

2.3 HARDWARE PLATFORMS

This section briefly describes two hardware platforms used to implement and analyse memory systems in this thesis: Field Programmable Gate Arrays (FPGAs) and Spatial Processors. Chapter 3 and 4 use FPGAs as target platforms, while Chapters 5 and 6 use Spatial Processors. Although the platforms are different, concepts regarding the design of the memory system are of a more generic nature, making it is possible to combine the knowledge obtained by the analysis of memory systems on these platforms.

2.3.1 *Field Programmable Gate Arrays (FPGAs)*

FPGAs are hardware platforms that can be repurposed after manufacturing. This is possible because the building block of FPGAs are Look Up Tables (LUTs) and an interconnect network. LUTs are used to implement functionality by providing the expected output for any given inputs. More complex behaviour is achieved by linking together LUTs using the interconnections network. The information about the routing of the interconnection network is itself stored in LUTs. This structure allows this hardware platform to be reconfigured after manufacturing, for the desired purpose, only by updating the the values in LUTs. FPGAs are used for two main purposes: hardware emulation, the original purpose for which they have been designed, and, more recently, as accelerators.

Hardware design has historically been specified using two Hardware Description Languages: VHDL and Verilog. These

are low level languages and give the hardware designer the ability to specify the hardware in a fine-grained manner. The synthesis process implements a hardware design, from an HDL specification to a target platform. If FPGAs are used as target platform, the output of the synthesis describes the configuration of the chip, defining what values need to be stored in each LUT. The freedom of specification provided by HDL comes, however, at the cost of lower productivity. Therefore, a lot of research is invested in defining and using languages with higher level of abstraction for hardware design. The result of this effort is a new synthesis process, called High Level Synthesis, that translates a specification given in an high level language to HDL.

Aside from LUTs, that guarantee the reconfigurability of these platforms, FPGAs usually contain other components, among which Block RAMs (BRAMs). These are blocks of memories similar to SRAMs, as they do not require the refresh of the stored data. There are usually several small independent BRAMs distributed across an FPGA chip. BRAMs can be linked to each other through the interconnection network to implement memories with higher capacity, and to the rest of the logic on the FPGA. This makes FPGAs ideal platforms for studying the design of a memory system, as they enable the design of memory controllers - using LUTs, different memory structures - changing the interconnections between BRAMs, and application logic - again, using LUTs.

In Chapters 3 and 4 FPGAs are used for implementing, validating and benchmarking memory system designs. Specifically, we use a Maxeler platform.

Maxeler builds FPGA boards for High Performance Computing using chips from Xilinx and Intel/Altera, the two major FPGA vendors. It uses an High Level Synthesis (HLS) language, called MaxJ, to describe the hardware. MaxJ adopts the dataflow programming paradigm, where an application is described as a directed graph: each node represents an operation on the data, while the edges represent the flow of data. During the computation, the data is streamed through the FPGA, and the operations are directly applied on the stream. The FPGA board features its own high capacity DRAM, which can be used to store application data. However, the latency of this memory is relatively

high (typical for off-chip DRAM) and even, with multi-channel implementations, the off-chip DRAM bandwidth is limited.

As a programming language, MaxJ is based on Java. It contains datatypes and operations useful to describe the dataflow graph of an application. From a MaxJ description, the Maxeler framework generates a dataflow graph that is then translated to a hardware description language (HDL). Finally, using third party tools, the HDL is synthesized and the bitstream required to program the FPGA is generated.

Chapters 3 and 4 show how to implement an on-chip cache on Maxeler platforms, aiming to maximize data reuse and minimize access to off-chip DRAM.

2.3.2 *Spatial Processor*

A *spatial processor* architecture consists of a set of physically distributed PEs with dedicated control units linked using an on-chip interconnect. The operations that need to be performed by an algorithm are mapped on the PEs, which compute in a fine-grained pipeline fashion. There are different kinds of spatial architectures, with one possible classification shown in Figure 2.5[59]. FPGAs are an example of spatially programmed architectures in which the PEs implement basic logic operations, and hence are classified as Logic-Grained. To change the functionality of a Logic-Grained architecture, the hardware design needs to be modified and re-synthesized. Instruction-Grained spatial architectures are, instead, programmable at instruction-level, and their PEs implement simplified ALUs. The functionality of an *Instruction-Grained* spatial accelerator can change by modifying the sequence of instructions it executes. The advantage of using *Instruction-Grained* over *Logic-Grained* programmable architectures lies in their higher computational density, which results in a higher operational frequency and lower power consumption[59]. The *Instruction-Grained* class is itself composed of architectures having Centralized Control[77] , where a single control unit manages all the PEs, and Distributed Control, where each PE has a built-in control mechanism [7, 59, 62]. Intuitively, an architecture with distributed control is more scalable and has a simpler interconnection network. Chapters 5 and 6 are target the

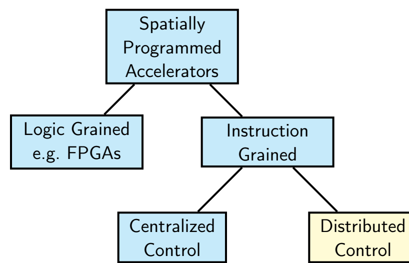


Figure 2.5: Spatial Architectures classification [59].

design of memory sytems for *distributed control* Spatial Processor architectures.

3 A PARALLEL SOFTWARE CACHE

One approach to address applications' demand for increased bandwidth is to re-think existing memory systems. Newly emerging technologies [34, 36] hold promise, but their large-scale integration depends on the processor vendors and is, therefore, rather slow. A more viable solution is to develop parallel memories, which could provide an immediate memory bandwidth increase as large as the number of parallel lanes. While this proposal sounds straightforward in theory, many challenges emerge when designing and/or implementing such memories in practice [88]. Efficient data writes, reading the data with a minimum number of accesses and maximum parallelism, and actually using such memories in real applications are only three of these challenges. In this chapter, we show how these challenges can be addressed in a systematic manner. Thus, this chapter addresses **RQ1**: What is a feasible design for a configurable hardware parallel memory?

> This chapter is based on:
> C. B. Ciobanu, **G. Stramondo**, C. de Laat, and A. L. Varbanescu *"MAX-PolyMem: High-Bandwidth Polymorphic Parallel Memories for DFEs"* [13], in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops*.

3.1 INTRODUCTION

To address the challenges relative to **RQ1**, we propose PolyMem, a Polymorphic Parallel Memory. We envision PolyMem as a high-bandwidth, two-dimensional (2D) memory which is *used to cache performance-critical data right on the FPGA chip*, making use of the existing distributed memory banks (the BRAMs). We chose a 2D address space for PolyMem to allow the programmers to easily place data structures such as vectors and matrices in this smart buffer, thus decreasing the need for complex index computation typically needed for a traditional, linear access

memory. Furthermore, using polymorphism, PolyMem not only delivers high performance for the most common two-dimensional access patterns (such as rows, columns, rectangles, or diagonals), but it also enables combining several such patterns in the same application. Finally, by supporting customization of capacity, bandwidth, number of read/write ports, and different parallel access patterns, PolyMem allows the user to configure the parallel memory to fit his/her application.

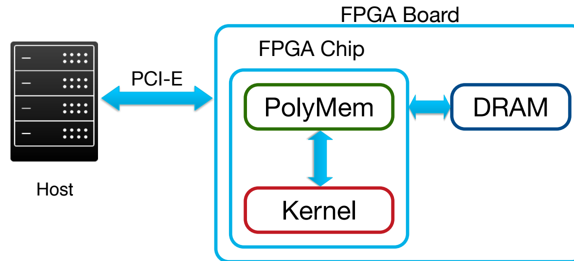


Figure 3.1: Envisioned system organization using PolyMemas a parallel cache.

Figure 3.1 depicts the envisioned system architecture. The FPGA board, featuring its own high-capacity DRAM memory, is connected to the host CPU through a PCI Express link. PolyMem acts like a high-bandwidth, 2D parallel software cache, able to feed an on-chip application kernel with multiple data elements every clock cycle.

PolyMem is inspired by existing research on the Polymorphic Register File (PRF) [17, 65]. While the PRF was designed as a runtime customizable register file for Single Instruction, Multiple Data (SIMD) co-processors, PolyMem is tailored for FPGA accelerators for High Performance Computing (HPC), which require high bandwidth but do not necessarily implement full-blown SIMD co-processors and their corresponding instruction sets on the reconfigurable fabric. We have selected FPGAs as our target for three reasons: (1) FPGAs are increasingly used for HPC acceleration due to their high energy efficiency and large amount of on-chip computational resources, (2) FPGAs enable PolyMem to be reconfigured depending on the current workload, and (3) current FPGAs feature relatively large amounts of on-chip, dis-

tributed, independent memories – i.e., the BRAM blocks – that can be used as parallel memory banks.

To enable a quick design and benefit from a high-level programming abstraction, our first prototype of PolyMem, called MAX-PolyMem[1], is implemented using Maxeler's platform and their MaxJ programming model [47]. This choice further enables us to easily integrate this parallel memory into Maxeler applications[2]. To thoroughly test the properties and limitations of MAX-PolyMem, we further propose a multi-dimensional Design Space Exploration (DSE) approach, where the capacity, number of lanes, and number of read ports for each PolyMem scheme are empirically evaluated. Our results show that (1) MAX-PolyMem can utilize the entire capacity of on-chip BRAMs, allowing the instantiation of a 4MB parallel memory on the Maxeler Vectis Data Flow Engine (DFE); (2) the maximum bandwidth delivered by the MaxJ design exceeds 32GB/s at a clock frequency of up to 202MHz, and (3) we are able to utilize all the available BRAMs with reasonable logic utilization.

Finally, to determine whether any unexpected bandwidth limitations occur when using MAX-PolyMem in practice, we have designed and implemented the STREAM benchmark [48, 79], which measures the bandwidth of different in-memory array operations. Using the COPY component of STREAM, we measured the bandwidth of a polymorphic memory with 1 read and 1 write port, and found that we achieve over 99% of the calculated peak performance.

In summary, the contributions of this work are the following:

- We introduce PolyMem, a Polymorphic Parallel Memory built using BRAMs as a high-throughput software-cache for FPGAs;

- We present MAX-PolyMem, the first prototype implementation of PolyMem for Maxeler's Data Flow Engines. We further analyze the productivity of MaxJ for our implementation: we quantify it through a combined metric (lines of

1 We use PolyMem to denote the *Polymorphic Memory design*, and MAX-PolyMem as the Maxeler-based implementation.
2 This integration work is beyond the scope of this paper.

Table 3.1: The PRF memory access schemes

PRF Access Scheme	Available Access Patterns
ReO (Rectangle Only)	Rectangle
ReRo (Rectangle, Row)	Rectangle, Row, Main and secondary Diagonals
ReCo (Rectangle, Column)	Rectangle, Column, Main and secondary Diagonals
RoCo (Row, Column)	Row, Column, Rectangle
ReTr (Rectangle, Transposed Rectangle)	Rectangle, Transposed Rectangle

code and development time), and qualify it through a set of lessons learned;

- We perform a DSE analysis to show how MAX-PolyMem scales with the number of lanes (up to 32), capacity (up to 4MB), clock frequency (up to 202MHz), and peak bandwidth (above 32GB/s).
- We design a MaxJ framework for the STREAM benchmark; we further implement and synthesize the STREAM-Copy application, and use it to benchmark the actual, achievable MAX-PolyMem bandwidth in practice.

3.1.1 *The Polymorphic Register File*

A PRF is a parameterizable register file, which can be logically reorganized by the programmer or a runtime system to support multiple register dimensions and sizes simultaneously [17]. The simultaneous support for multiple conflict-free access patterns, called *multiview*, is crucial, providing flexibility and improved performance for target applications. The polymorphism aspect refers to the support for adjusting the sizes and shapes of the registers at runtime. In Table 3.1, each *multiview* scheme (ReRo, ReCo, RoCo and ReTr) supports a combination of at least two conflict-free access patterns.

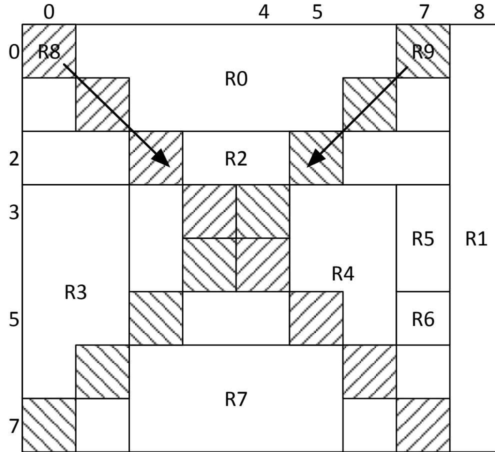


Figure 3.2: PolyMem supported access patterns.

In this work, we reuse the PRF conflict-free parallel storage techniques and patterns, as well as the polymorphism idea to design PolyMem. Figure 3.2 illustrates the set of access patterns supported by the PRF and, ultimately, by PolyMem. In this example, a 2D logical address space of 8×9 elements contains 10 memory Regions (R), each with different size and location: matrix, transposed matrix, row, column, main and secondary diagonals. Assuming a hardware implementation with eight memory banks, each of these regions can be read using one (R1-R9) or several (R0) parallel accesses.

By design, the PRF optimizes the memory throughput for a set of predefined memory access patterns. For PolyMem, we consider $p \times q$ memory modules and the five parallel access schemes presented in Table 3.1. Each scheme supports dense, conflict-free access to $p \cdot q$ elements[3]. When implemented in reconfigurable technology, PolyMem allows application-driven customization: its capacity, number of read/write ports, and the number of lanes can be set pre-runtime (or even at runtime using partial reconfiguration), to best support the application needs.

In summary, PolyMem uses the technology developed for the PRF to build a parallel memory (Figure 3.2) for three reasons: (1) it provides a generic, out-of-the-box solution to implement

3 In this work, we will use "×" to refer to a 2D matrix, and "·" to denote multiplication.

a parallel memory, thus avoiding error-prone, time-consuming custom memory design; (2) it can be customized for the application at hand; (3) its multi-view property allows 2D arrays to be distributed across several BRAMs, enabling runtime parallel data access using *multiple, different "shapes"* without the need for hardware reconfiguration (see Table 3.1). Effectively, with the PRF-based PolyMem, programmers can assume a parallel memory and focus on algorithm optimizations rather than complex data transformations or low-level details.

3.2 DESIGN AND IMPLEMENTATION

In this section, we briefly present our approach for designing PolyMem to fit a given application, and further dive into the implementation of MAX-PolyMem. This implementation is open source, and is available online at [61].

3.2.1 *End-to-end design*

A great advantage of PolyMem is its ability to be configured to fit the needs of given applications. A configuration consists of a storage capacity **C** (e.g., 512KB), distributed in $p \times q$ memory lanes, a PRF access scheme, and the number of read ports. The access scheme enables support for up to four parallel-access patterns (out of the six supported - see 3.1.1), each of which is a dense access to $p \cdot q$ elements. To customize PolyMem for a given application, we start from the application memory access pattern, for which we find the *optimal parallel access schedule* - i.e., the best sequence of parallel accesses to the application data - for each potential configuration (scheme, capacity, lanes). To determine the optimal schedule we formulate the problem as a set covering problem [37], using Integer Linear Programming (ILP) for the search itself. We finally select the best configuration based on two metrics: speedup and efficiency. More details on this process are presented in [71].

3.2.2 *From PolyMem to MAX-PolyMem*

Figure 3.3 shows a diagram describing MAX-PolyMem, our MaxJ PolyMem implementation; we further refer to blocks in this Figure in **bold** and to signals with a spaced-out `font`.

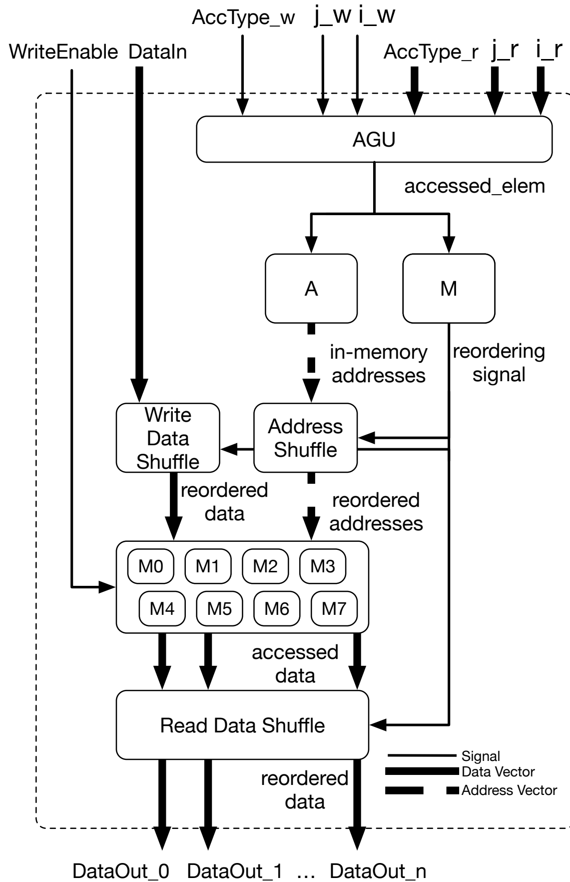


Figure 3.3: The Block-diagram of our MAX-PolyMem Implementation.

Because PolyMem behaves as a 2D memory, parallel application accesses are made using two coordinates, `(i,j)`, and the shape of a parallel access, `AccType`. `DataIn` and `DataOut` represent the data which is written to and read from MAX-PolyMem.

The core of MAX-PolyMem's design consists of a 2D array of memories ($p \times q$ BRAMs). These are used to store the data in a distributed manner. In Figure 3.3, eight such memories are

Table 3.2: Productivity analysis.

Module/Feature	Effort (days)	LOC
AGU	2	194
A	3	292
Shuffle	10	335
M	4	399
Memory banks	3	242
Inv Shuffle	4	346
Multiple Read Ports	1	127

illustrated (**M0-M7**); these are the **Memory Banks**, also called memory modules. The number of banks defines the number of data elements which are read/written in parallel per data port, referred to as lanes.

Based on the `(i,j)` coordinates and the requested access type `AccType`, the **AGU** expands the parallel access in its individual components by computing the coordinates of all the accessed elements ($p \times q$ addresses in total). This operation is performed for the write port and for each read port, so that one write access and one read access for each read port can happen independently at the same time.

The Module Assignment Function (MAF) is a mathematical function that maps each element in the 2D address space to one of the **Memory Banks**. The MAF guarantees conflict-free access to the supported access patterns. In this work, we use the five MAFs listed in Table 3.1 and described in detail in [17]. **M** implements all the MAFs supported by our design and outputs the select signal in the three types of **Shuffles**: **Read Data Shuffle**, **Address Shuffle**, and **Write Data Shuffle**. The Shuffles are implemented using full crossbars and are used to reorder input and output data according to the MAF used. The Addressing Function **A** computes, for each accessed element, the intra-memory bank address. The **Read/Write Data Shuffle** and **Address Shuffle** reorder the data elements and their corresponding intra-memory bank addresses (generated by the **A**) so that each memory bank receives the correct address and input data.

For each access to PolyMem, the input signals and data flow through all the blocks of the design in Figure 3.3, top to bottom. Both the `DataIn` and `DataOut` are arranged in our predefined order (left to right, top to bottom) to ensure consistency between reads and writes. When writing to PolyMem, the **Memory Banks** store each input element into the assigned memory module at the corresponding intra-memory module address. More specifically, the input data - `DataIn` - is written in the memory locations identified by **A** and **M**, after they have been reordered by the **Write Data Shuffle**. During a read access, the output of the **Memory Banks**, containing the accessed data, is reordered by the **Read Data Shuffle**. If the `WriteEnable` signal is low the `DataIn` elements are ignored. Simultaneous reads and writes are supported because of the independent read and write ports, and our design supports multiple read ports.

We note that our design is implemented using two types of **Shuffles**. Given a reordering signal, the regular **Shuffle** reorders the elements, while the **Inverse Shuffle**, with the same reordering signal, restores the initial order. In this design, therefore, the **Write Data Shuffle** is implemented using an inverse **Shuffle**, while the **Read Data Shuffle** is implemented using a regular **Shuffle**.

3.2.3 *Productivity Analysis*

One of the reasons for using Maxeler's platform for this work was the alleged ease-of-use of the MaxJ toolchain. We reflect here on our development process and analyze, qualitatively and quantitatively, the productivity of MaxJ.

The development process started by implementing each module in Figure 3.3 in isolation. Table 3.2 illustrates the implementation effort (in days) taken by each module, as well as the required LOC (lines of code). In our experience, Maxeler's toolchain does enable fast prototyping: it takes little effort to have a simple kernel running on a Maxeler board, due to the Java-like language and the integrated behavioural simulator.

Once all kernels were available, we created a modular multi-kernel design, using a custom manager to connect the different modules. This approach helped testing and debugging. We found

that the lack of a graphical representation of the blocks in a design forces the developer to programmatically link the modules, a time-consuming and error-prone process; futhermore, some of the toolchain errors are not documented: we had problems with the PCI-express interface, the file management in the IDE, and several simulator crashing/hanging instances. With all these issues, the integration took 5 days.

We further explored the trade-off between modularity and performance: we implemented a *fused*, single-kernel implementation (which took 7 days) and compared the two versions. We found that the modular version consumes twice as many resources, mainly due to the additional inter-kernel communication infrastructure.

Aiming to further optimize the code, we ran into the real challenge of most HLS approaches: low-level details of the implementation are hidden within layers of abstractions and tools, and low-level optimizations are difficult to integrate.

Overall, we find that Maxeler's toolchain is an asset during the first development stages of an application because (1) MaxJ's HLS approach hides most of the complexity of hardware design, (2) the behavioral simulator from the MaxIDE saves time during implementation and debugging, and (3) the design can be written with very few lines of code and it is easily readable. On the downside, more documentation and tool support are needed to fine-tune and optimize non-trivial applications. Moreover, the integration of multiple kernels into a single design is complex due to the lack of visualization tools, and MaxJ makes it difficult to fine-tune low-level behavior.

3.3 DESIGN SPACE EXPLORATION AND RESULTS

We analyze the performance of MAX-PolyMem through DSE, reporting memory bandwidth (see 3.3.2) and resource utilization (see 3.3.3).

3.3.1 *Design Space Exploration setup*

For this study, we have selected three relevant parameters for the design space exploration, listed and explained in Table 3.3. For all

experiments in this paper we use a Maxeler Vectis board that uses a Xilinx Virtex-6 SX475T FPGA[4] featuring 475k logic cells and 4MB of on-chip BRAMs. All our experiments configure PolyMem for a data width of 64 bits. Our design is easily configurable: a simple configuration file sets, at compile time, the required DSE parameters. We collected information regarding the FPGA resource usage and the clock frequency for each configuration. We have further computed the maximum read and write bandwidth that can be achieved. We validate each design with a simple read-/write cycle: the host fills MAX-PolyMem with unique numerical values, and then reads them back using parallel accesses. The remainder of this section focuses on the detailed analysis of these results.

Table 3.3: PolyMem Design Space Exploration Parameters.

DSE Parameter	Values	Explanation / *Affected blocks*
Total Size [KB]	512, 1024, 2048, 4096	*The number and capacity of each Memory Bank*
Number of lanes (p $\times q$)	8 (2×4), 16 (2×8)	Number of data elements delivered for each port per clock cycle. *Affects each block of the design.*
Number of Read Ports	1, 2, 3, 4	Number of independent data blocks, $p \cdot q$ elements each, which can be read in each clock cycle. *Affects the aggregate PolyMem bandwidth and the number and capacity of each Memory Bank*

3.3.2 *Memory Performance*

In its role as a parallel memory, the most important performance metric for MAX-PolyMem is memory bandwidth. We compute the maximum bandwidth assuming all accesses use the full width of the memory. The main parameters influencing the bandwidth

4 Xilinx Virtex-6 Family Overview:
http://xilinx.com/support/documentation/data_sheets/ds150.pdf

Table 3.4: MAX-PolyMem Maximum Clock Frequencies [MHz].

Size	512KB						1024KB						2048KB				4096KB	
Scheme / Lanes	8				16		8				16		8		16		8	16
Read Ports	1	2	3	4	1	2	1	2	3	4	1	2	1	2	1	2	1	1
ReO	202	160	139	123	185	100	160	123	102	79	144	109	127	86	127	87	95	95
ReRo	195	166	131	123	168	100	163	125	102	77	140	109	120	87	120	80	98	91
ReCo	196	155	131	122	157	100	163	121	107	81	156	122	124	78	124	79	93	93
RoCo	194	150	146	122	161	100	173	135	114	86	145	109	122	90	122	84	88	91
ReTr	193	158	134	137	159	112	155	121	102	77	146	122	116	81	114	77	102	102

are: design clock frequency, which varies depending on the MAX-PolyMem parameters (see Table 3.4), the number of lanes, and the number of read ports.

Table 3.4 lists the maximum clock frequencies achieved by our designs. The highest frequency, 202MHz, is achieved by the 512KB, 8-lane, single read port ReO design. For the multi-view schemes, the highest clock frequency is 196MHz for the 512KB, 8-lane, single read port ReCo configuration. The minimum clock frequency is 77MHz.

Figure 3.4 presents the maximum achievable bandwidth per single port, which is also the write bandwidth of our designs. The peak write bandwidth for the 16-lane configurations exceeds 22GB/s for the 512KB, 16-lane, ReO configuration. For the multiview schemes, the maximum achieved bandwidth is 20GB/s for the ReRo configuration. Moreover, we note that single-port bandwidth scales linearly when doubling number of memory banks from 8 to 16.

Figure 3.5 illustrates the maximum read bandwidth when increasing the number of read ports. The peak bandwidth is 32GB/s achieved by the 512KB, 8-lane, 4-port ReTr scheme. For the 8-lane configurations, we observe good bandwidth scaling when doubling the number of ports from 1 to 2 ports, and diminishing returns for the 3- and 4-port configurations. If the number of lanes is increased to 16, having 2 read ports does not significantly increase the bandwidth. We also note that bandwidth is reduced if the number of lanes and ports is kept constant, but the capacity of PolyMem is increased. This is most likely due to the additional pressure put on the synthesis tools to place and route all the additional BRAMs.

Please note that for the applications that utilize the read and write ports simultaneously, the total total delivered PolyMem

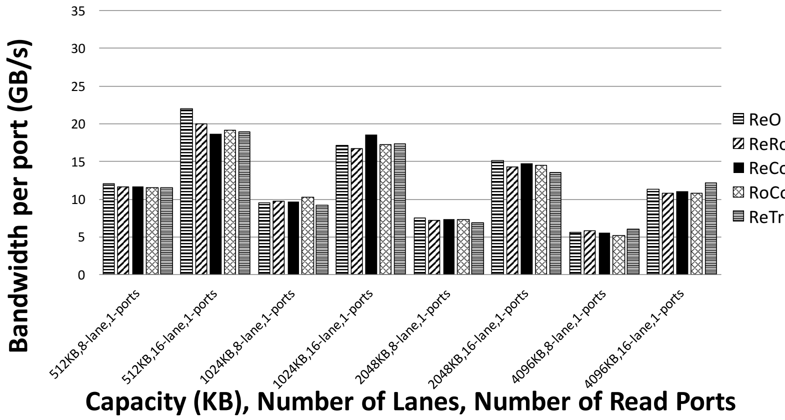


Figure 3.4: Write bandwidth.

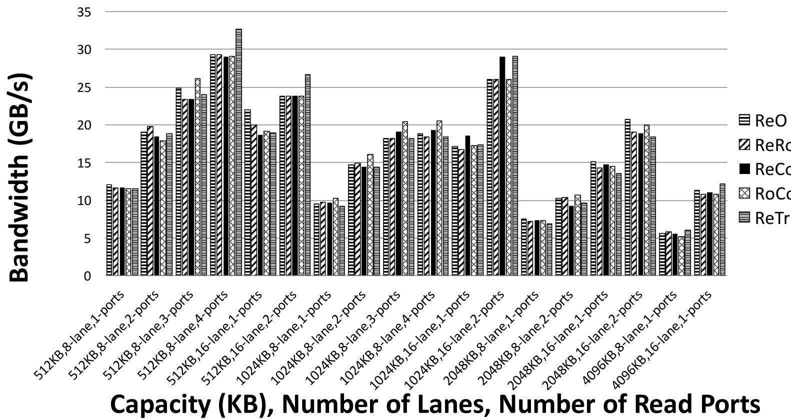


Figure 3.5: Read bandwidth (aggregated).

data rate is the sum of the bandwidth delivered by all individual read and write ports.

3.3.3 *Resource utilization*

We continue by analyzing the Maxeler Vectis DFE synthesis results in terms of resource utilization. Specifically, we investigated logic, LUT, and BRAM utilization (Figures 3.6, 3.7, and 3.8, respectively).

The results indicate that when increasing the PolyMem capacity but keeping the number of lanes and ports constant, there

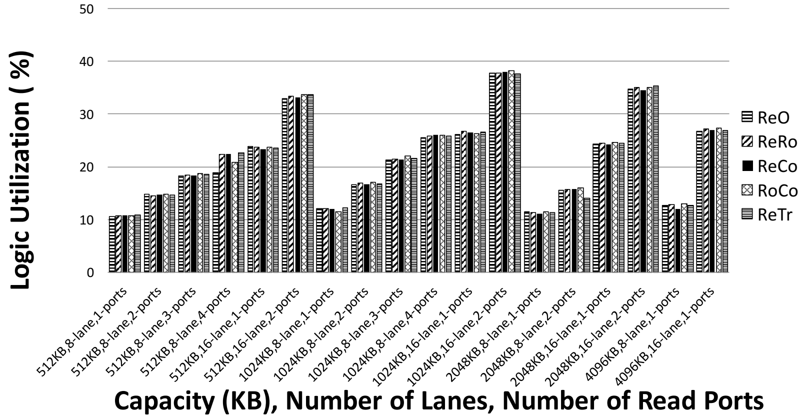


Figure 3.6: Logic utilization.

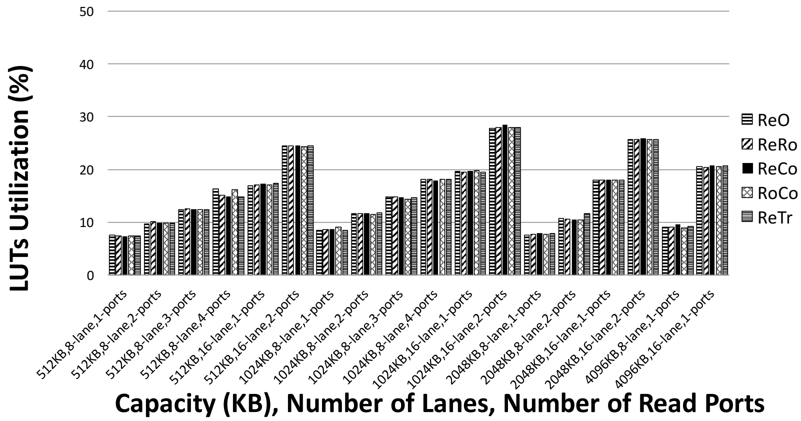


Figure 3.7: LUT Utilization.

is little to no increase in logic utilization for any of the target memory schemes. For example, MAX-PolyMem with 8 lanes and a single read port, the logic utilization varies between 10.58% for a 512KB, ReO configuration to 13.05% for the 4096KB featuring the RoCo scheme. However, the increasing the number of read ports does increase the logic utilization: for the ReRo, 512KB, 8 lane configuration, the logic utilization doubles from 10.78% for the single port case to 22.34% for the 4-port PolyMem, mostly due to the read crossbars replication.

When doubling the lanes count from 8 to 16, we observe a supra-linear logic utilization increase. For example, for the 512KB,

single read port, ReRo PolyMem, the logic utilization increases from 10.78% to 23.73%. This can be attributed to the quadratic increase in resource used by the full crossbar in relation to the number of lanes [17].

Figure 3.7 illustrates the LUTs utilization. We observe here similar trends to the logic utilization described above, with the LUTs utilization varying between 7% and 28%.

Finally, Figure 3.8 illustrates the BRAM utilization, which varies from around 16% for a 512KB, 8-lane, 1-read port PolyMem up to 97% for a 2MB, 16-lane, 2-read ports PolyMem. As expected, the memory scheme has no influence on the amount of BRAMs used. Increasing the PolyMem capacity and increasing the number of PolyMem lanes and read ports leads to an increased BRAM utilization. For example, for the single read port ReRo, 512KB design, the 8-lane configuration utilizes 16.07% of the BRAMs, the 16-lane PolyMem uses 19.31% and the 8-lane, dual read port configuration uses 29.04% of the BRAMs. This behavior is the expected one since increasing the number of read ports involved duplicating data in BRAMs.

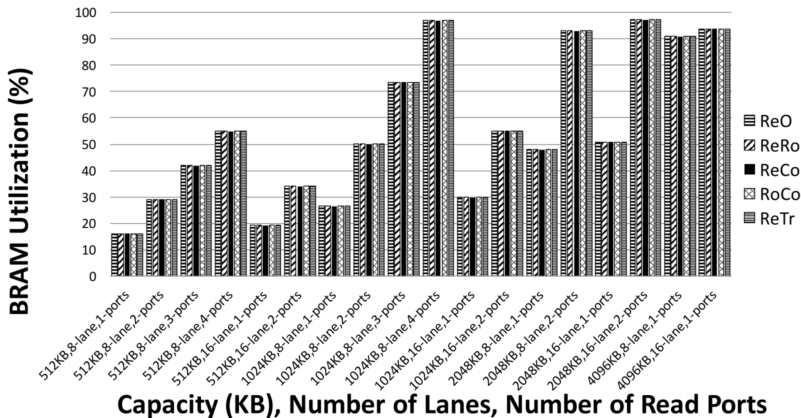


Figure 3.8: BRAM Utilization.

In summary, we make the following observations:

- MAX-PolyMem is able to utilize the entire capacity of on-chip BRAMs, allowing the instantiation of a 4MB parallel memory on the Maxeler Vectis DFE while keeping the logic utilization under 38% and LUTs usage under 28%;

- Supra-linear logic and LUTs increase when doubling the number of lanes;
- MAX-PolyMem delivers up to 22GB/s write bandwidth and up to 32GB/s aggregated read bandwidth using up to 4 read ports, at a clock frequency of up to 202MHz.

3.4 STREAM-COPY: BANDWIDTH BENCHMARKING

This section focuses on the empirical evaluation of PolyMem's performance in practice. We aim to demonstrate that our implementation has a measured throughput in line with the estimated values presented in Section 3.3. For this analysis, we have used the STREAM benchmark [48, 79], a well-known tool for memory bandwidth estimation in modern computing systems.

The STREAM benchmark contains four applications: Copy, Scale, Sum, and Triad. The benchmark uses three vectors - A, B and C - in all its applications. The Copy application performs a vector copy operation $c(i) = a(i)$, which involves one read and one write for each element copied. The Scale application performs the scaling of a vector and stores its result in another vector $a(i) = q \cdot b(i)$; thus performing two memory accesses (a read and a write) and one floating point multiplication per element processed. The Sum application performs the sum of two vectors, $a(i) = b(i) + c(i)$, featuring two read, one write, and a floating point addition per element. Finally, the Triad application is a combination of the Scale and Sum, $a(i) = b(i) + q \cdot c(i)$, thus featuring two reads, one write, and the two floating point operations, a multiplication and an addition.

To use STREAM for the assessment of MAX-PolyMem, we must design the STREAM framework using Maxeler's toolchain and MAX-PolyMem. A high-level view of our design, which is open-source and available online [75], is presented in Figure 4.4. The host is connected through the PCI-e to our STREAM design, and starts the computation by sending the `Vector Sizes` and `Mode` parameters to define the behavior of the **Controller**. The **Controller** generates the write and read signals for MAX-PolyMemand selects the correct input for MAX-PolyMem's write port by driving the the two **MUXs**. The signals `Wi`, `Wj` and `Wshape`

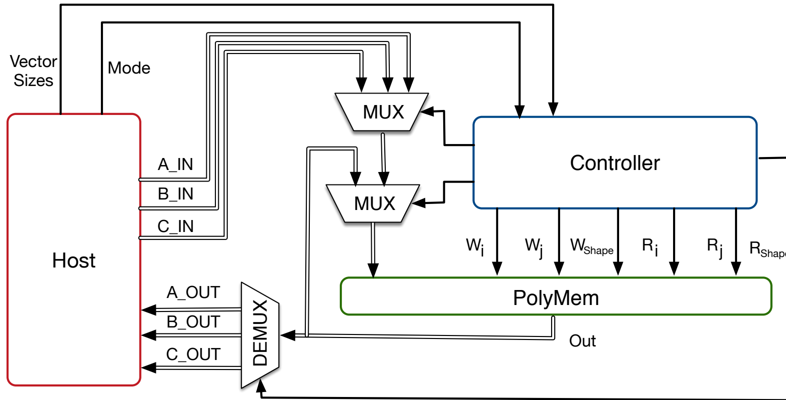


Figure 3.9: The implementation of the STREAM benchmark for MAX-PolyMem. All transfers between host (the CPU) and PolyMem (on the FPGA) are done via the PCIe link.

and `Ri`, `Rj`, `Rshape` signals identify the write/read locations for the elements to be stored in/retrieved from **PolyMem**. Lastly, using the **DEMUX**, the controller selects the right output stream (from `A_OUT`, `B_OUT`, `C_OUT`) to correctly retrieve the data from the **PolyMem**.

All the results we present further in this section are obtained using the STREM-Copy application, which enables us to measure the *achieved aggregated bandwidth for a design with 1 read and 1 write port*, and report them using the standard reporting of the STREAM benchmark itself.

For measurements, we split the design in three separate stages: *Load*, *Offload*, and *Copy*. The current stage is specified by the host through the `Mode` signal. During the *Load* stage, the three vectors (A, B and C) are loaded into PolyMem, which is split in three (equally-sized) regions, to store each of these arrays. The **Controller** makes sure each array is written in its own space. In the second stage, *Copy*, the elements contained in vector A are copied in vector C. The parallel read and write operations can happen in simultaneously: the controller selects the feedback loop from the output port of **PolyMem**. The delay introduced by the read operation (i.e., its latency), is taken into account by our design, ensuring that the controller's inputs to PolyMem are correctly aligned with the output of the parallel memory.

The required delay applied on the output data is 14 clock cycles (estimated by Maxeler's tools). Finally, in the last stage, *Offload*, the host retrieves the data from the PolyMem(A, B and C) using three separate streams.

Each of these stages is ran in isolation, orchestrated by the host. The use of blocking calls ensures the separation between stages, also enabling a clear separation between the stages' execution times. Our focus is on the accurate measurement of the *Copy* stage, which represents the actual STREAM-Copy application, and is used to benchmark MAX-PolyMem's bandwidth.

For our experiments, we synthesized this design using a PolyMem with 8 lanes ($p \times q = 2 \times 4$). Because we access data in rows only, we have used the RoCo scheme. All arrays use 64-bit elements. The maximum allocated size for each array is $170 \times 512 \times 8$ bytes, which amounts to around 700KB. This limitation is due to the STREAM design, using 2 read ports, which translates to 2MB of storage effectively available. However, because STREAM-Copy only uses one read port, the design was optimized at synthesis - i.e., its complexity was reduced to that of a single read port design. Thus, we were able to synthesize this STREAM-Copy design with one read and one write ports at 120MHz, just 2 MHz lower than the maximum clock frequency for a 2048KB configuration with a single read port listed in Table 3.4.

Figure 3.10 shows the combined read/write throughput we measured with the Copy application, without the data transfers - which happen in separate stages and whose execution time does not contribute to our measurements. The reported data are obtained by measuring 1000 runs of the copy operation, to ensure sufficient measurement resolution and to limit the impact of the minimum overhead of the host-FPGA signal communication. This minimum overhead is, according to our dedicated measurements, around 300ns, and interferes with any measurements of applications with comparable runtimes. This effect is visible on the left side of the graph of Figure 3.10, before the memory reaches its sustained bandwidth.

As for the theoretical peak of this memory, we have 2 ports, each with 8 memory lanes, each lane being 64-bit wide. Thus, the aggregated (read + write) theoretical bandwidth of this copy-

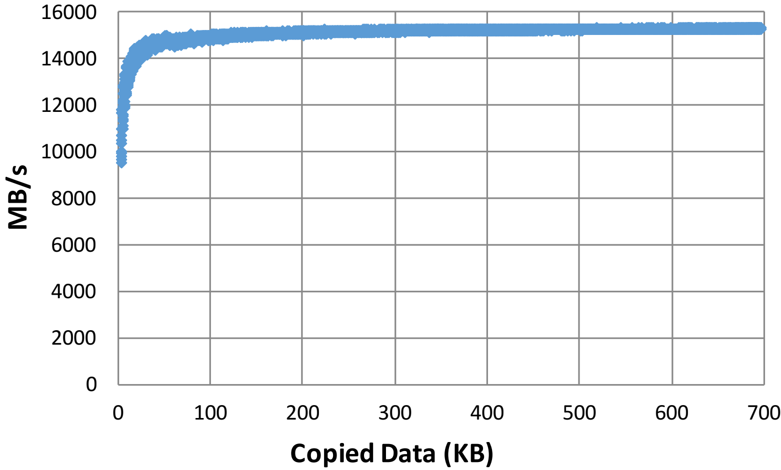


Figure 3.10: Copy bandwidth (aggregated).

ing operation is $2 \times 8 \times 8 \times 120 = 15360$ MB/s. The maximum measured throughput we obtained from our STREAM Copy benchmark is 15301 MB/s, which represents more than 99% of the theoretical bandwidth. We conclude that our STREAM implementation validates the peak performance of MAX-PolyMem, demonstrating very little overhead when using the memory in practice.

3.5 RELATED WORK

Building a memory hierarchy for FPGA kernels is recognized as a difficult, error-prone task [1, 64]. For example, [1, 10, 28, 51, 87] focus on the design of generic, traditional caches. By comparison, our work proposes a parallel, polymorphic memory which *acts as a caching mechanism* between the DRAM and the processing logic; instead of supporting placement and replacement policies, our memory is configured for the application at hand, and it is directly accessible for reading and writing.

Application-specific caches have also been investigated [10, 63, 86], though none of these are polymorphic or parallel. Of special interest to this work is [56], where the authors demonstrate why and how different caches can be instantiated for specific data structures with different access patterns. PolyMem starts from

a similar idea, but, benefiting from its multi-view, polymorphic design, it improves on it by using a single large memory for all these data structures.

Many of PolyMem's advantages arise from its PRF-based design [17], which is more flexible and performs better than alternative memory systems [21, 41, 58, 60]; its high performance in scientific applications has also been proven for practical applications [15, 18, 65].

In summary, compared to previous work on enabling easy-to-use memory hierarchies and/or caching mechanisms for FPGAs, PolyMem proposes a PRF-based design that, to the best of our knowledge, is the first to support polymorphic parallel accesses through a single, multi-view, application-specific software cache. Moreover, MAX-PolyMem is the first prototype of a parallel software cache written entirely in MaxJ, and targeted at Maxeler DFEs.

3.6 SUMMARY

This chapter focused on the performance improvement of FPGA-accelerated applications through increased memory-system parallelism. In this context, PolyMem is an easily configurable, 2D multi-bank software caching mechanism which provides both performance - by combining BRAMs, multi-view parallel data accesses, and concurrent read and write operations - with the flexibility of read/write operations. Due to its multi-view parallel accesses, PolyMem enables applications with dense and/or sparse memory access patterns to benefit from memory-system parallelism.

We have implemented our prototype on the Maxeler platform, using MaxJ. Thus, MAX-PolyMem is a high bandwidth parallel caching mechanism, fully implemented in MaxJ, for Maxeler's DFEs. As such, it can be directly integrated in other MaxJ designs which require a parallel memory, as proven by our STREAM implementation.

We have tested the limits of our prototype on the Maxeler Vectis DFE board. Our results show that the design can utilize the entire capacity of on-chip BRAMs, and parallel memories up to 4MB, featuring up to 16-lanes, and/or supporting up to 4

read ports are feasible. Assuming dense access patterns, MAX-PolyMem's estimated peak bandwidth is up to 22GB/s for writes and above 32GB/s for reads. Finally, using a MaxJ implementation of the STREAM-Copy benchmark, we were able to confirm that MAX-PolyMem can achieve more than 99% of the estimated aggregated (read+write) peak bandwidth in practice.

Additional improvements that can be made to the implementation of our design are left for future work. The further development of a proof-of-concept, systematic method to use MAX-PolyMem for more complex applications is also desirable. In fact, our ultimate goal is to provide an HLS toolchain that can analyze applications, determine the requirements and configurations for the most suitable PolyMem based configurations, and enable the seamless integration of these high-bandwidth caching mechanisms with the target applications. A first step towards this goal is presented in the next chapter.

4 APPLICATION-CENTRIC PARALLEL MEMORIES

Memory bandwidth is a critical performance factor for many applications and architectures. Intuitively, a parallel memory could be a good solution for any bandwidth-limited application, yet building *application-centric custom parallel memories* remains a challenge. In this chapter, we present a comprehensive approach to tackle this challenge, and demonstrate how to systematically design and implement application-centric parallel memories. We build this approach "on top" of the parallel memory design presented in Chapter 3. Specifically, our approach (1) analyzes the application memory access traces to extract parallel accesses, (2) configures the parallel memory for maximum performance, and (3) builds the actual application-centric memory system. We further provide a simple performance prediction model for the constructed memory system.

We evaluate our approach with two sets of experiments. First, we demonstrate how our parallel memories provide performance benefits for a broad range of memory access patterns. Second, we prove the feasibility of our approach *and* validate our performance model by implementing and benchmarking the designed parallel memories using FPGA hardware and a sparse version of the STREAM benchmark. Our results demonstrate that such a systematic approach is feasible, thus addressing **RQ2**: Can we define and implement a systematic, application-centric method to configure a hardware parallel memory?

This chapter is based on:
Giulio Stramondo, Cătălin Bogdan Ciobanu, Ana Lucia Varbanescu, and Cees de Laat *"Towards application-centric parallel memories"* [73], in *European Conference on Parallel Processing*.

Giulio Stramondo, Cătălin Bogdan Ciobanu, Cees de Laat, and Ana Lucia Varbanescu *"Designing and building application centric parallel memories"* [72], in *Concurrency and Computation: Practice and Experience*.

4.1 INTRODUCTION

Many heterogeneous systems are currently based on massively parallel accelerators (e.g., GPUs), built for compute-heavy applications. Although these accelerators offer significantly larger memory bandwidth than regular CPUs, many kernels using them are bandwidth-bound. This means that, in practice, more bandwidth is needed for most of our applications. Therefore, our work addresses the need for increased bandwidth by enabling more parallelism in the memory system. In other words, for bandwidth-bound applications, this work demonstrates how to build heterogeneous platforms using parallel-memory accelerators.

Designing and/or implementing application-specific parallel memories is non-trivial [4]. Writing the data efficiently, reading the data with a minimum number of accesses and maximum parallelism, and using such memories in real applications are significant challenges. In this paper, we describe our comprehensive approach to designing, building, and using parallel-memory application-specific accelerators. Our parallel memory is designed based on PolyMem [14], a polymorphic parallel memory model with a given set of predefined parallel access patterns. Our approach follows four stages: (1) analyze the memory access trace of the given application to extract parallel memory accesses (Section 4.2), (2) configure PolyMem to maximize the performance of the memory system for the given application (Sections 4.3.1- 4.3.3), (3) compute the (close-to-)optimal mapping and scheduling of application concurrent memory accesses to PolyMem accesses (Figure 4.1, Section 4.3.4), and (4) implement the actual accelerator (using MAX-PolyMem), also embedding its management into the host code (Section 4.5.1).

The performance of our accelerators is assessed using two metrics: speed-up against an equivalent accelerator with a sequential memory, and efficiency. Based on a simple, yet accurate model that estimates the bandwidth of the resulting memory system. Using this estimate and benchmarking data, we could further estimate the overall performance gain of the application using the newly built heterogeneous system.

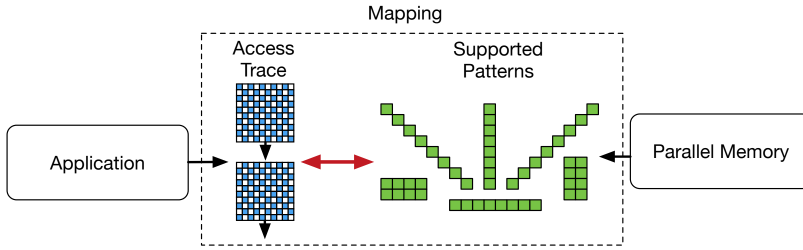


Figure 4.1: Customizing parallel memories. Our research focuses on the mapping of the access trace from the application to the parallel access patterns of the parallel memory.

We validate our approach using 10 Sparse STREAM instances: the original (dense) and 9 variants with various sparsity levels (Section 4.5). We demonstrate how our method enables a seamless analysis and implementation of 10 accelerators in hardware (using a Maxeler FPGA board). Finally, using real benchmarking data from the PolyMem-based heterogeneous systems, we validate our performance model.

In summary, our contribution in this work is four-fold:

- We present a methodology to analyze and transform application access traces into a sequence of parallel memory accesses.
- We provide a systematic approach to optimally configure a polymorphic parallel memory (e.g., PolyMem) and schedule the set of memory accesses to maximize the performance of the resulting memory system.
- We define and validate a model that predicts the performance of our parallel-memory system.
- We present empirical evidence that the designs generated using our approach can be implemented in hardware as parallel-memory accelerators, delivering the predicted performance.

4.2 PRELIMINARIES AND TERMINOLOGY

In this section we remind the reader the terminology and basic definitions necessary to understand the remainder of this chapter.

4.2.1 *Parallel Memories*

A parallel memory enables the access to multiple elements in parallel. It can be realized by combining several *sequential memories*.The number of sequential memories used in implementing such a parallel memory represents the maximum number of elements that can be read in parallel - also called *width of the parallel memory*. The amount of data that can be stored in the (parallel) memory is called *memory capacity*.

A specific element contained in a PM is identified by its *location*, a combination of a *memory module identifier* (to specify which sequential memory hosts the data) and an *in-memory address* (to specify where within that memory the element is stored). We call this pair *the parallel memory location* of the data element. Formally, thus, $loc(A[I]) = (m_k, addr), k = [0..M)$, where $A[I]$ represents an element of the application - see Section 4.2.2., m_k is the memory module identifier, M is the width of the PM, and $addr$ is the in-memory address.

Our approach focuses on *non-redundant* parallel memories, i.e., memories use a one-to-one mapping between the coordinate of an element in the application space and a memory location, can use the full capacity of all the memory resources available, and data consistency is guaranteed by avoiding data replication. However, these parallel memories restrict the possible parallel accesses: only elements stored in different memories can be accessed in parallel (see Section (see Chapter 2).

4.2.2 *The Application*

We use the term *application* to refer to the entity using the PM to read/write data - e.g., a hardware element directly connected to the PM, or a software application interfaced with the PM.
As seen in Chapter 2, we consider the following terminology:

- The data of an application is stored in an N-dimensions array A;
- $A[I] = A[i_0][i_1]...[i_{N-1}]$ are the coordinates of element $I = (i_0, i_1, ..., i_{N-1})$ in the *application space*.
- A *memory access* is a read/write memory operation.

- A *concurrent access* is a concurrent set of memory accesses, $A[I_j], j = 1..P$, which the application can perform concurrently.
- An *application memory access trace* is a temporally-ordered series of *concurrent accesses*.
- A *parallel memory access* is an access to multiple data elements which actually happens in parallel.

Ideally, to maximize the performance of an application, any concurrent access should be a parallel access, happening in one memory cycle. However, when the size of a concurrent access (P) is larger than the width of the PM (M), a scheduling step is required, to schedule all P accesses on the M memories. Our goal is to systematically minimize the number of parallel accesses for each concurrent access in the application trace. We do so by tweaking both the memory configuration and the scheduling itself.

Tweaking the memory configuration

To specify a M-wide parallel access to array A – stored in the PM –, one can explicitly enumerate M addresses ($A[I_0]...A[I_{M-1}]$), or use an *access pattern*. The access pattern is expressed as a M-wide set of N-dimensional offsets - i.e.,$\{(o_{0,0}, o_{0,1}, ..., o_{0,N-1}) - (o_{M-1,0}, o_{M-1,1}, ..., o_{M-1,N-1})\}$. Using a reference address - i.e. $A[I]$ - and the access pattern makes it possible to derive all M addresses to be accessed. For example, for a 4-wide access (M=4) in a 2D array (N=2), where the accesses are at the N,E,S,W elements, the access pattern is $\{(-1, 0), (0, -1), (1, 0), (0, 1)\}$. When combining the pattern with a reference address - e.g., (4,4) - we obtain a set of M element coordinates - e.g, $\{(3, 4), (4, 3), (5, 4), (4, 5)\}$. We call the operation of instantiating a memory access pattern into a set of addresses based on a reference address *resolving the pattern*. In Section 4.3.2 we will use the function *resolve_pattern(p,a)* - where p is an access pattern and a is a reference address - to indicate this operation.

As defined in Chapter 2, in definition 3, a set of memory accesses $A[I_0]..A[I_{Q-1}]$ form a parallel memory access iff the set constitutes a conflict-free parallel access.

To map the access to an element in application space to a parallel access in PM space, we need to define a mapping function that guarantees M-wide conflict free accesses. Determining the function to use is a key challenge in defining a custom parallel memory.

Definition 4 (Memory Mapping Function) *The Memory Mapping Function (MMF) maps an application memory access to its parallel memory location.*

$$MMF : (A[I], M, D[I]) \rightarrow (m_k, addr_k), k = [0..M)$$

where $I = (i_0, i_1, ..., i_{N-1})$ *are the coordinates of the access in the application space,* M *is the width of the parallel memory, and* $D[I]$ *are the sizes of each dimension of the application space array.*

We note that due to the restriction that only conflict-free accesses can be parallel accesses, there is a limited set of access patterns that a parallel memory can support. These patterns are an immediate consequence of the MMF.

A PM configuration is the pair (MMF, C), where MMF is a mapping function and C is the capacity of the PM. Customizing a parallel memory entails finding, for a given application, the configuration that minimizes the number of parallel accesses to the PM.

In the remainder of this paper we focus on a methodology to configure a custom parallel memory with the right M, C, and MMF for a given application (see Section 4.3 and further).

Scheduling concurrent accesses

Once the parallel memory configuration is known, the transformation between the application concurrent accesses and the memory parallel accesses is necessary. We call this transformation *scheduling*, and note it can be static - i.e., computed pre-runtime, per concurrent access - or dynamic - i.e., computed at runtime. In this work, we assume static scheduling is possible, and the actual schedule is an outcome of our methodology (see Section 4.3 and further).

4.3 SCHEDULING AN APPLICATION ACCESS TRACE TO A PM

In this section we describe two approaches for scheduling an application access trace using a set of PM parallel access patterns. The first one finds an optimal solution to this problem - the minimum number of PM accesses that cover the application access trace - using ILP. The second one proposes an alternative to ILP, in the form of a heuristic method which trades-off optimality for speed. Finally, we end this section with an overview of our full approach towards application-centric parallel memories and a simple predictive model to calculate the performance of the resulting memory system.

4.3.1 *The set covering problem*

We express the problem of scheduling an application access trace onto a set of PM accesses as a particular instance of the *set covering* NP-complete problem [81].

Definition 5 (Set Covering[81]) *Given a universe* $\mathbb{U}$ *of n elements, a collection of sets* $\mathbb{S} = \{S_1, ..., S_k\}$*, with* $S_i \subseteq \mathbb{U}$*, and a cost function* $c : \mathbb{S} \rightarrow Q+$*, find a minimum-cost subset of* $\mathbb{S}$ *that covers all elements of* $\mathbb{U}$*.*

The set cover can be formulated as an integer program:

$$\begin{aligned} \text{minimize} \quad & \sum_{S_i \in \mathbb{S}} \mathrm{c}(S_i) \cdot x_{S_i} \\ \text{subject to} \quad & \sum_{S_i : e \in S_i} x_{S_i} \geq 1, \quad e \in \mathbb{U}. \end{aligned}$$

In this formulation, $x_{S_i} = \{0, 1\}$ is a variable indicating if set S_i is part of the solution, $\mathrm{c}(S_i)$ is the cost of set S_i, and the solution is constrained to have for each element $e \in \mathbb{U}$ at least one set $S_i : e \in S_i$.

4.3.2 *From Concurrent Accesses to Set Covering*

An optimal schedule of an application access trace on a set of PM parallel accesses can be found by reducing this problem to a set

covering one, and leveraging the ILP formulation discussed in the previous section. Although an application access trace contains a list of application concurrent accesses, we schedule each of those separately. For every application concurrent access, the universe $\mathbb{U}$ is formed by all accesses. From the PM predefined parallel access patterns, we define $\mathbb{S}$ as the collection of all possible parallel accesses in PM (see algorithm 1). Finally, the solution obtained using an ILP solver, $\mathbb{S}_{min}, \mathbb{S}_{min} \subseteq \mathbb{S}$, is a list of sets which optimally cover the concurrent accesses, and will be converted back into a sequence of parallel memory accesses.

Algorithm 1 Generation of the Collection of Sets

1: $\mathbb{S} \leftarrow \emptyset$
2: $\mathbb{A} \leftarrow$ {all application elements}
3: $\mathbb{U} \leftarrow$ {all accessed elements}
4: $\mathbb{P} \leftarrow$ {PM parallel access patterns}
5: **for** $p \in \mathbb{P}$ **do**
6:     **for** $a \in \mathbb{A}$ **do**
7:         $pa \leftarrow \text{resolve_pattern}(p, a)$.
8:         $S_{pa} \leftarrow pa \cap \mathbb{U}$.
9:         $\mathbb{S} \leftarrow \mathbb{S} \cup S_{pa}$
10:     **end for**
11: **end for**
12: **return** $\mathbb{S}$.

Algorithm 1 shows how to generate $\mathbb{S}$, from which the minimal coverage will be extracted. Set $\mathbb{P}$ contains the list of PM conflict-free accesses patterns, and it is obtained from the PM configuration. Set $\mathbb{A}$ contains the coordinates of the application data. Each pair of an application element and an access pattern (i.e., elements from $\mathbb{A}$ and $\mathbb{P}$, respectively) is resolved into a set of coordinates of application elements, *pa*, by *resolve_pattern* (see Section 4.2.1); To map our problem to the ILP formulation above we need to guarantee that the union of the collection of subsets in $\mathbb{S}$ is equal to the universe $\mathbb{U}$. This is done by removing the elements that are not being accessed in the concurrent access -i.e. the elements in $\mathbb{A}$ but not in $\mathbb{U}$- from the parallel access *pa*. The elements of $\mathbb{S}$ will be all these S_{pa} sets, for which it holds that $\bigcup_{S_{pa} \in \mathbb{S}} S_{pa} = \mathbb{U}$.

To solve our original problem, we are interested in finding the minimum collection of sets $\mathbb{S}_{min}$ such that $\bigcup_{S \in \mathbb{S}_{min}} S = \mathbb{U}$ and $\mathbb{S}_{min} \subseteq \mathbb{S}$, so the cost function will be defined as $c(S_{pa}) = 1, \forall S_{pa} \in \mathbb{S}$. Once $\mathbb{S}, \mathbb{U}, c$ are defined, an ILP solver can be used to compute $\mathbb{S}_{min}$ - the minimum collection of sets that covers the universe $\mathbb{U}$.

4.3.3 *An Heuristic Approach*

As our preliminary results show that ILP is a major bottleneck in our system, speed-wise, we also investigate the possibility to offer an alternative to the ILP formulation for solving the scheduling problem. Therefore, we have designed and implemented a heuristic approach, based on a greedy algorithm (see Algorithm 2). Our heuristic is based on [81], and the solution is guaranteed to be within an harmonic factor from the optimal solution (extracted with the ILP approach).

Algorithm 2 Heuristic Application Trace Scheduling

1: $\mathbb{U} \leftarrow \{$all accessed elements$\}$
2: $\mathbb{S} \leftarrow \{$possible parallel accesses$\}$
3: $\mathbb{S}_h \leftarrow \varnothing$
4: $E \leftarrow \mathbb{U}$
5: **while** $E \neq \varnothing$ **do**
6: Find $S_{pa} \in \mathbb{S}$ s.t. $|E \backslash S_{pa}|$ is minimum.
7: $\mathbb{S}_h \leftarrow \mathbb{S}_h \cup S_{pa}$.
8: $E \leftarrow E \backslash S_{pa}$
9: **end while**
10: **return** $\mathbb{S}_h$.

Algorithm 2 shows our heuristic approach. E is a set used to keep track of the elements still to be covered with a parallel access, and it is initialized with $\mathbb{U}$, the set containing all the elements in the concurrent access. $\mathbb{S}$ contains all parallel accesses from $\mathbb{A}$ for a given PM configuration (Algorithm 1, Section 4.3.2). In each iteration, the parallel access $S_{pa} \in \mathbb{S}$, which contains the maximum number of elements that still needs to be covered, is added to the solution, and the elements covered by S_{pa} are removed from E. Once all the elements in the application concur-

rent access have been covered, the algorithm returns the set of parallel access $\mathbb{S}_h$ containing the solution.

4.3.4 *The Complete Approach*

Our complete approach is presented in Figure 4.2. We start from the *Application Access Trace*, a description of the concurrent accesses in the application, discussed in detail in Section 4.2.2. We test different parallel memory configuration by providing different *Configuration Files* to our *Memory Simulator*. Each *Configuration File* contains details regarding mapping scheme, number of parallel lanes and capacity of the parallel memory. The *Memory Simulator* produces all the available parallel accesses, compatible with the given parallel memory *Configuration File*, that cover elements contained in the *Application Access Trace*. The set of parallel accesses is then given as input to our ILP or Heuristic solver - implemented as described in Sections 4.3.2 and 4.3.3. The *Solver* selects the minimum number of parallel accesses that fully cover the elements in the *Application Access Trace*, thus producing a *Schedule* of parallel memory accesses. The *Schedule* can then directly be used in the hardware implementation of the application parallel memory.

An important side-effect of our approach is that the information contained in the schedule can further be used to accurately estimate the performance of the generated memory system. Thus, to calculate the achievable average bandwidth of the memory system for the given access trace, we can "penalize" the theoretical bandwidth (i.e., assuming that all lanes are fully used) by our efficiency metric: $BW_{real} = BW_{peak} \times Efficiency = (Frequency * Bitwidth * Lanes) \times \frac{N_{seq}}{N_{elements}}$. *Frequency* is the frequency the PM is operating at, *Bitwidth* is the size of each element stored in the PM and *Lanes* represents the amount of elements that can be accessed in parallel; N_{seq} is the number of required sequential accesses and $N_{elements}$ is the total number of elements accessed by the PM using a *Schedule*.

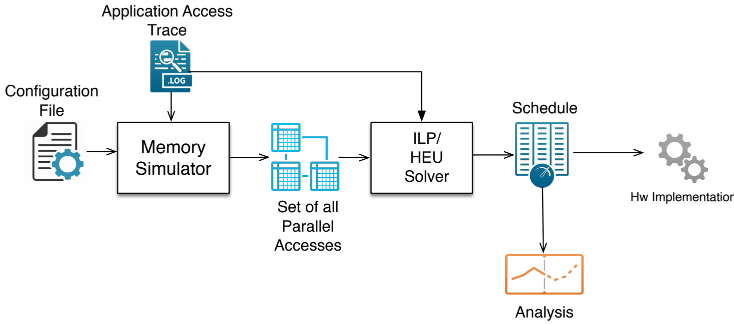


Figure 4.2: An overview of our complete approach.

4.4 EVALUATION

This section describes a *statistical* analysis, based on simulation results, of the potential benefits of PMs for different types of applications, characterized by their memory access patterns. It further compares the solutions obtained by our heuristic against the optimal solutions produced by the ILP algorithm (see Section 4.3).

4.4.1 *Experiment Setup*

To empirically demonstrate the potential of parallel memories to improve bandwidth and, ultimately, provide speed-up over non-parallel solutions even for non-dense memory access patterns, we propose an experiment where we test the PM for a large number of synthetic memory access patterns. We assume that the capacity of the PM is sufficient to contain the application data. For each pattern, we measure both the performance gain and the efficiency of using PMs. This experiment also enables us compare the two algorithms for scheduling a memory access trace (see Sections 4.3.3 and 4.3.2).

SYNTHETIC APPLICATION CONCURRENT ACCESSES The set of concurrent accesses - *strided* - is generated assuming an 8x8 data structure and using three parameters: offset, number of reads, and number of skips. The pattern is generated alternating series of reads and series of skips. The offset defines the number of elements to skip from the element [0][0]. The entire set of

synthetic concurrent accesses has been generated using 8 by 8 patterns with offset varying from 0 to 7, number of reads varying from 1 to 8, number of skips from 1 to 8. This resulted in a total of 512 application access traces.

PM CONFIGURATIONS The Memory Mapping Functions (MMFs) used in the PM configurations guarantee conflict free access to the following 2D patterns(see Figure 4.1): Rectangle, Diagonal, Secondary Diagonal, Row, Column, and Transposed Rectangle. We assumed a memory capacity sufficient to store all application data, and experimented with a PM width, M, from 2 to 8 (M=8 is sufficient to allow full rows/columns/diagonals to be read from our synthetic concurrent accesses) - and all combinations of the PRF access patterns. In total, we tested 448 different PM configurations.

EVALUATION METRICS We introduce two metrics to evaluate how an application benefits from a parallel memory: speed-up and efficiency.

Speed-up is a measure of the performance gain from using a custom parallel memory, defined as: $Speed-up = \frac{N_{seq}}{N_{par}}$. N_{seq} refers to the number of accesses required using a sequential memory - i.e. equal to the number of elements in the application concurrent access - N_{par} is the number of parallel memory accesses, obtained using the algorithms in Section 4.3.

Efficiency is a measure of the "wasted accesses" when using a custom parallel memory, defined as: $Efficiency = \frac{N_{seq}}{N_{elem}}$. N_{elem} is the total number of elements accesses by the PM and it is equal to $N_{par} \times M$, where M is width of PM. We note that efficiency is an indirect measure of the overhead of a parallel memory for a sparse access, and can be correlated with the power efficiency of the memory system.

4.4.2 *Results*

We have scheduled all 512 synthetic concurrent accesses (Section 4.4.1) on all 448 memory configurations (Section 4.4.1) using both the algorithms proposed in Section 4.3 - ILP and heuristic.

To determine whether the custom parallel memories are successful in improving the performance of different applications, we analyze speed-up; to determine whether the heuristic algorithm can be used as a replacement of the ILP-based solution, we analyze the observed trade-off between the optimality (by ILP method) and speed (by the heuristic method).

speed-up Figure 4.3a shows the speed-up results, grouped per PM-width. We make the following observations:

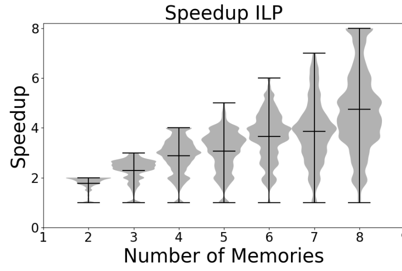
- The bottom parts of the plots, indicating low speed-ups, are very narrow, showing that only very few concurrent accesses did not benefit from using PM. This is correlated to the sparsity of the memory accesses in the concurrent access and the fact that the parallel access patterns we used only allow dense parallel accesses.

- The top parts of the violins, corresponding to high speed-ups, are also narrow, indicating that only few concurrent accesses can gain maximum speed-up. Moreover, the figure also shows that for odd numbers of memories (3,5,7), the occurrence of close-to maximum and maximum speed-up is very rare: in fact, 1-5 patterns, at most, reach the maximum).

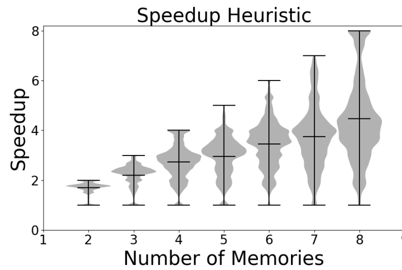
- The majority of the concurrent accesses lies in between those two extremes, showing that they gain significant speed-up by using PM, but that it is not possible to fully utilize the all memory banks.

We note that our efficiency results (not included due to space limitations) show a similar picture: few (concurrent access, PM configuration) pairs gain maximum or minimum efficiency, while the average efficiency varies between 0.8 for 2 memories and 0.58 for 8 memories.

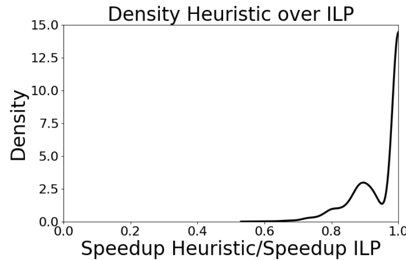
ilp versus heuristic Figure 4.3b presents the speed-up results for all concurrent accesses and memory configurations, using the heuristic algorithm instead of the ILP. From the figure, there is little difference in the distribution of the speed-ups: few configurations at the bottom and at the top, and most configura-



(a) Violin Plot showing the results obtained using the ILP algorithm.



(b) Violin Plot showing the results obtained using the HEU algorithm.



(c) Density plot of the ratio between the ILP speed-up and the HEU speed-up.

Figure 4.3: Evaluation of the ILP and Heuristic (HEU) results.

tions in the middle. On average, the heuristic approach gives a speed-up of 0,05% below the optimum computed with ILP.

To see in how many cases the difference between the ILP and heuristic approaches is significant, we compute the ratio $\frac{Speedup_{heu}}{Speedup_{ILP}}$ and plot the density distribution of this ratio in Figure 4.3c. In

the large majority of cases, the speed-ups are similar, with a loss is less than 15%; the worst result obtained by the heuristic is 53% of the optimal speed-up for one single configuration. These results indicate that the heuristic algorithm is acceptable as a replacement of the ILP when quick estimation is required.

4.5 EXPERIMENTS AND RESULTS

We evaluate the feasibility and performance of our approach by designing and implementing 10 parallel-memory accelerators on an FPGA-based system. We use a Maxeler Vectis board, equipped with a Xilinx Virtex-6 SX475T FPGA[1] featuring 475k logic cells and 4MB of on-chip BRAMs.

4.5.1 *MAX-PolyMem*

Our parallel memory is based on PolyMem, a design inspired by the polymorphic register file [16]. The hardware implementations and performance analysis presented in this section are all based on the Maxeler version of PolyMem, MAX-PolyMem, presented in Chapter 3. We include here a brief reminder of MAX-PolyMem's design and implementation characteristics.

PolyMem is a non-redundant parallel memory, using multiple lanes to enable parallel data access to bi-dimensional data structures, and a specialized hardware module that enables parallelism for multiple access patterns. For example, an 8-lane PolyMem allows reading/writing 8 elements at a time from/to a 2D memory. The access shapes supported by PolyMem, defined as bi-dimensional shapes, are Row, Column, Rectangle, Transposed Rectangle, Main Diagonal, and Secondary Diagonal. Due to its multi-view design [16], PolyMem supports several *access schemes*, i.e, it can perform memory operations with different access patterns without reconfiguration:

- ReO: Rectangle.
- ReRo: Rectangle, Row, Diagonal, Sec. Diagonal.

1 Xilinx Virtex-6 Family Overview:
http://xilinx.com/support/documentation/data_sheets/ds150.pdf

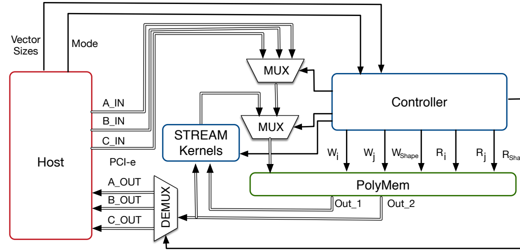


Figure 4.4: The implementation of the STREAM benchmark for MAX-PolyMem (figure updated from [14]). All transfers between host (the CPU) and PolyMem (on the FPGA) are done via the PCIe link.

- ReCo: Rectangle, Column, Diagonal, Sec. Diagonal.
- RoCo: Row, Column, Rectangle.
- ReTr: Rectangle, Transposed Rectangle.

4.5.2 *Sparse STREAM*

To prove the feasibility of our approach, from application access traces to hardware, we adapt the STREAM benchmark [48, 79], a well-known tool for memory bandwidth estimation in modern computing systems, to support sparse accesses.

The original STREAM benchmark uses three *dense* vectors - A, B and C - and proposes four kernels: Copy (C=A), Scale ($A = q \cdot B$), Sum ($A = B + C$), and Triad ($A = B + q \cdot C$).

We have designed a version of STREAM for MAX-PolyMem [14]. A high-level view of our design[2], is presented in Figure 4.4.

However, the original STREAM does not challenge our approach because it uses dense, regular accesses. We therefore propose Sparse STREAM, an adaptation of STREAM which allows 2D arrays and configurable sparse accesses. Table 4.1 presents 10 possible variants of Sparse STREAM, labeled based on their read access density. The main difference between these variants is its number of sequential accesses, N_{seq}.

We apply our methodology for each variant. Thus, for each variant, we obtain the (close-to-) optimal schedule per access scheme. The schedule is characterized by the number of parallel

2 STREAM for MAX-PolyMem is open-source and available online [75].

Pattern description			ReRo Scheme				RoCo Scheme				Selected
Density	Pattern	N_{seq}	N_{par}	$N_{elements}$	Speed-up	Efficiency	N_{par}	$N_{elements}$	Speed-up	Efficiency	Scheme
20	RR________RR____	17408	4369	34952	3.98	49.81	4369	34952	3.98	49.81	ReRo
25	R___R___R___R___	21760	10880	87040	2.00	25.00	2816	22528	7.73	96.59	RoCo
33	R__R__R__R__R__R	29013	3724	29792	7.79	97.39	9671	77368	3.00	37.50	ReRo
40	RRRR____RRRR____	34816	8687	69496	4.01	50.10	8687	69496	4.01	50.10	ReRo
50	R_R_R_R_R_R_R_R_	43519	10880	87040	4.00	50.00	5504	44032	7.91	98.83	RoCo
60	RRRRRR____RRRRRR	52224	8821	70568	5.92	74.01	8821	70568	5.92	74.01	ReRo
66	RR_RR_RR_RR_RR_R	58026	7350	58800	7.89	98.68	9710	77680	5.98	74.70	ReRo
75	RRR_RRR_RRR_RRR_	65279	10880	87040	6.00	75.00	8192	65536	7.97	99.61	RoCo
80	RRRRRRRR__RRRRRR	69632	8806	70448	7.91	98.84	8806	70448	7.91	98.84	ReRo
100	RRRRRRRRRRRRRRRR	87040	10880	87040	8.00	100.00	10880	87040	8.00	100.00	ReRo

Table 4.1: The 10 variants of the STREAM benchmark and the predicted performance of the calculated schedules for two schemes (ReRo and RoCo). The other schemes are omitted because they are not competitive for these patterns. In the patterns, only the `R` elements need to be read.

accesses N_{par}, and the total number of accessed elements $N_{elements}$ (Section 4.3), from which we calculate speed-up and efficiency per access scheme. We present these results for two schemes (namely, ReRo and RoCo) in Table 4.1. We select the best performing to test in hardware.

The final step in our approach is the translation from a schedule to a hardware implementation of our parallel-memory accelerator. The key challenge is to enable the controller (see Figure 4.4) to orchestrate the parallel memory operations based on the given schedule. Our current prototype stores the schedule, which contains information regarding the required sequence of parallel accesses (coordinates, shape, and mask), in an on-chip `Schedule memory`.

4.5.3 *Results*

We have implemented all 10 STREAM variants in hardware by configuring MAX-PolyMem, for each test-case, with a memory of 261120 elements (i.e., the maximum capacity available fitting the arrays A, B, C and the `schedule memory`), and the best scheme (see Table 4.1). We have measured the performance of each STREAM component and compared it against our bandwidth estimation.

We measure the bandwidth of our 10 Sparse STREAM kernels (average over 10000 runs)*[3]. The results - predicted vs. measured - are presented in Figure 4.5. We make the following observations:

- Our performance model (see Section 4.3) accurately predicts the performance of the memory system (below 1% error in most cases).
- For 6 out of the 9 sparse STREAM variants, we can achieve close to optimal speed-up due to our parallel memory being multi-view and polymorphic.
- For S-25, S-50, and S-75, the performance gain versus choosing the alternative scheme used in this experiment is, according to Table 4.1, of 70%, 50%, and 24%, respectively.
- Our STREAM PolyMem design uses only 25.98% of the logic available in the Vectis Maxeler board. More information regarding the resource usage is available in [14].

Overall, our experiments are successful: we demonstrated that the schedule generated by our approach can be used in real-hardware, and we showed that the measured performance is practically the same with the predicted one.

4.6 RELATED WORK

Research on using parallel memories to improve system memory bandwidth has started in the 70s, and remains of interest today. Parallel memories that use a set of predefined mapping functions to enable specifically shaped parallel accesses have improved to better support more shapes [29, 30, 42], multiple views, and polymorphic access [16]. Approaches that derive an application-specific mapping function [83, 88] have also emerged, constantly improving the efficiency and performance of the generated memory systems. The current version of this work uses a polymorphic parallel memory with fixed shapes, to which we add the novel analysis and configuration methodology.

3 The overhead of uploading/downloading the arrays to PolyMem is not included in these results.

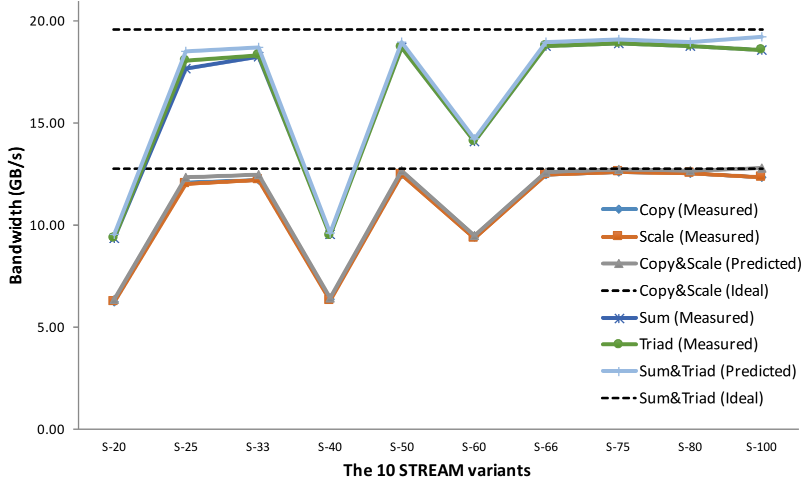


Figure 4.5: The performance results (measured, predicted, and ideal) for the 10 different variants of the STREAM benchmark. The horizontal lines indicate the theoretical bandwidth of MAX-PolyMem, configured with 8-byte data, 8 lanes, and 2 (for Copy and Scale) or 3 (for Sum or Triad) parallel operations. Running at 100MHz, MAX-PolyMem can reach up to 12.8GB/s for 1-operand benchmarks and up to 19.6GB/s for 2-operand benchmarks.

As for building such memories in hardware, a lot of research has been invested in building application-specific caches for FPGAs. Although successful, such research [10, 63, 86] does not (yet) address parallel and/or polymorphic memories. Our work fills this gap, by showing how to efficiently design a polymorphic, multi-view parallel memory embedded into an FPGA-based accelerator.

4.7 SUMMARY

Modern accelerators, currently embedded in heterogeneous systems, offer massive parallelism for compute-intensive applications, but often suffer from memory bandwidth limitations. Our work investigates the benefits of building accelerators with application-specific parallel memories as a solution to alleviate this bottleneck. Our approach is especially effective for applications with large sets of concurrent accesses.

In this chapter we proposed an end-to-end workflow which, given an application, analyzes the application access trace, configures and builds a custom non-redundant parallel memory (e.g., PolyMem), optimized for the data-intensive kernel of interest, generates our parallel-memory accelerator in hardware, and embeds it in the original host code.

We have empirically validated our approach using Sparse STREAM with 10 different access densities. We demonstrated that we can instantiate and benchmark all 10 designs in real hardware (i.e., a Maxeler system and the MAX-PolyMem version). Our experimental results demonstrate clear bandwidth gains, and closely match our model's predictions. To further perfect this method, more work is needed on the analysis of more applications. Three important steps need to be taken in this direction: (1) improve/automate the access traces extraction, (2) provide a more efficient integration of the parallel-memory accelerator into the host application, and (3) design an extension of the model towards accurate full-application performance prediction.

5 COMPUTE AND MEMORY SYSTEM CODESIGN

As we see with most modern accelerators, the growing performance gap between memories and processors effectively means the memory system often determines the overall performance and power consumption in silicon. One other solution (besides that presented in Chapter 4) to address the increasing demand in performance and energy efficiency of both embedded and high performance computing systems is thorough novel system architectures such as spatial processors.

The tight interdependency between the memory system and spatial processor architectures suggests that they can be codesigned. However, no effective methods and tools are available for this process: the only possible alternative, the state-of-the-art design methodologies for processor architecture, are ineffective for spatial processor architectures because they do not include the memory system.

In this chapter, we present μ-Genie, an automated framework for codesign-space exploration of spatial processor architecture and the memory system, starting from an application description in a high-level programming language. In addition, we propose a spatial processor architecture template that can be configured at design-time for optimal hardware implementation. Thus, this chapter addresses **RQ3**: Is there a systematic way to codesign efficient processing and memory systems from a given application?

This chapter is based on:
Giulio Stramondo, Manil Dev Gomony, Bartek Kozicki, Cees De Laat, and Ana Lucia Varbanescu *"μ-Genie: A Framework for Memory-Aware Custom Processor Architecture Co-Design Exploration"* [74], in *Digital System Design*.

5.1 INTRODUCTION

In modern embedded and high performance computing systems, there is increasing interest in *spatial processors*. These processing architectures consisting of physically distributed Processing Elements (PEs), and are more energy efficient than traditional general purpose processors and reconfigurable hardware accelerators. [3, 7, 9, 24, 59, 62, 69]. Despite the growing interest, we are still far from having the right solutions and automated tools for designing efficient, high-performance spatial processors. An important drawback of existing design solutions is their inability to take the memory system into account, despite compelling evidence that the memory system (both on-chip and off-chip) has become a dominant factor affecting the overall performance, power consumption, and silicon area usage [22, 57, 84]. In this work we argue that, given that the memory system and the processing elements in a spatial processor architecture are so tightly *interdependent*, they must be *codesigned* to efficiently use the available resources. A memory-aware design produces spatial architectures having bandwidth close to the bandwidth of the memory system, effectively reducing the instantiation of unrequired resources. Codesigning becomes especially important when considering the use of emerging memory technologies, such as MRAM, eDRAM, PCM, or RRAM [8] in spatial processors: they have higher integration density and lower power than SRAM, but also come with additional "quirks" (e.g., MRAM features different read and write latencies).

There are CAD tools [6, 19, 78] that reduce the increasing design complexity of typical application-specific processors. However, selecting an optimal spatial processor architecture, taking into account the various trade-offs in latency, power consumption, and area usage still requires extensive design-space exploration (DSE) and cannot be performed using the existing CAD tools. In addition, state-of-the-art design flows for application-specific processor DSE focus on processing elements optimization [26, 35, 38, 50], and do not include the memory system, as illustrated in Figure 5.1. Instead, co-optimization of the processor and the memory system (including emerging memories) is typically done

through the optimization of cache replacement policies [40, 44, 52, 76].

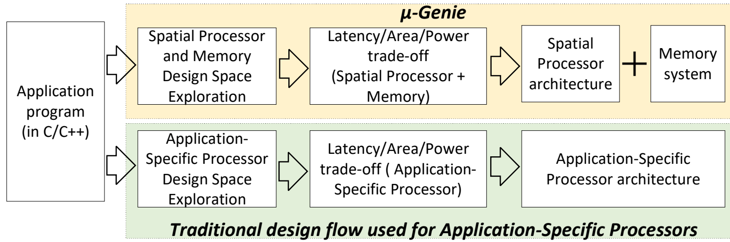


Figure 5.1: Difference between state-of-the-art design flow typically used for traditional application-specific processors and the proposed μ-Genie design flow for spatial processors.

This paper presents as main contribution μ-Genie (Section 5.2), an automated framework for *memory-aware spatial processor design-space exploration*. The framework presented in this work allows the user to **customize** the different building blocks to be used in the codesign of the spatial processor and memory system, **explore** different architectures automatically generated for a given application, and **estimate** area, power and latency of each one of the architectures.

This chapter highlights the following key contributions:

- Unprecedented configuration options: memory levels technologies (novel among similar tools), clock frequency (per memory level, also novel), different read/write latencies, and data-widths.

- A configurable PE architecture template (Section 5.6) that allows fast prototyping of spatial processor hardware.

- The Modified Interval Partitioning (MIP) algorithm (Section 5.4.3), that enables the memory-aware (co)Design Space Exploration (Section 5.5).

To demonstrate the capabilities of μ-Genie, we cover three case-studies, showing how a spatial processor can be designed for two different applications and many configurations, including those using MRAM or SRAM for the memory system, can be generated, analyzed, and compared (Section 6.2).

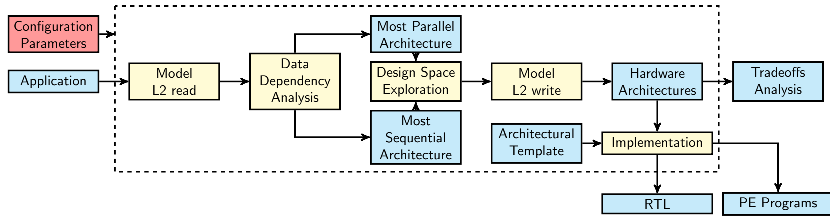


Figure 5.2: μ-Genie Framework.

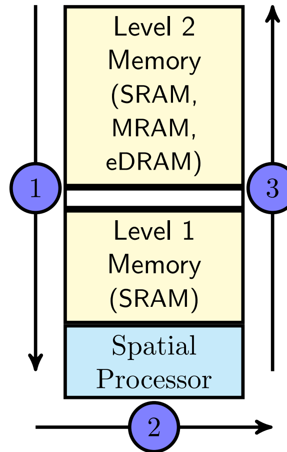


Figure 5.3: The system under analysis.

5.2 THE μ-GENIEFRAMEWORK

The μ-Genie framework, illustrated in Figure 5.2 takes two inputs, the *Configuration Parameters* - described in Section 5.3.2 - and an *Application* - detailed in Section 5.3.1, and automatically generates a set of hardware architectures, behaviorally equivalent to the input application. The generated hardware architectures can be realized as RTL implementations, using the *architectural templates* described in Section 5.6. The rest of this section provides a detailed analysis of the design and implementation of μ-Genie.

5.2.1 *Model of Execution*

The system architecture we assume in this work has two levels of memory and a spatial processor (Figure 5.3). Level 1 memory (L1M)[1], the first memory level, and the smaller one in size, uses

SRAM as it needs to be physically close to the processor for faster access. The second level - Level 2 memory (L2M)[1] - is larger in size and can be implemented using any memory technology (on-chip or off-chip), even with different access latency for read and write operations. Note that L2M can run at a different clock speed and different IO width than the processor and L1M. We assume a model of execution following the three steps, from Figure 5.3, shown by arrows representing the direction of data flow. Initially, all the required input data for the application are available in L2M. The input data is transferred to the processor, using L1M as intermediate storage (step 1), the data is processed and the results are temporarily stored in L1M (step 2), and, finally, the data from L1M is transferred back to L2M (step 3). The data transfer between L2M and L1M are handled by a Direct Memory Access (DMA) controller. Note that our model of execution performs the steps in a pipelined manner, hence only part of the data will be stored in L1M at any given time.

μ-Genie lets the user specify the parameters of the L2M through the *Configuration Parameters*. The L2M parameters are used to model the data transfer between L2M and L1M (see 5.2.2 and 5.4.1). The L2M model is used to compute arrival time of input elements in the L1M. The arrival time of the element in the L1M is then used by the Modified Interval Partitioning (MIP) algorithm - see 5.4.3, to produce spatial architectures having bandwidth close to the bandwidth of the L2M. This effectively reduces the instantiation of unrequired resources in both the L1M and the spatial processor. The L1M is composed of multiple banks having different depths. The number and depths of the banks composing the L1M is determined by the MIP as described in 5.5.1.

5.2.2 *The L2 Memory Model*

Because L2M has higher access latency compared to the L1M and spatial processor, we model the L2M assuming its data is accessed in bursts. A read or write burst access to the L2M is

1 These memories are not to be seen as caches; thus, no cache policies are needed: we schedule data movements at design time. This is why we call them "levels" instead of "layers", and we abbreviate them with L1M and L2M instead of L1 and L2.

controlled by a Direct Memory Access (DMA) controller, with the starting address and size of the burst given as input to the DMA. After an initial *setup latency*, the accessed elements are transferred in sequence from the start address to the end address, from L1M to L2M in case of a write, and from L2M to L1M in case of a read.

5.3 μ-GENIE: INPUTS

This section details the two inputs of μ-Genie: the *application* and the *Configuration Parameters*.

5.3.1 *Application*

The applications that can be used as input to μ-Genie are completely defined at compile time, having control-flow instructions not dependent on input data. Such applications enable the static extraction of data dependency information performed by the *Data Dependency Analysis* module (5.4.2). We currently support C/C++ applications. However, as the framework uses the LLVM[2] Intermediate Representation [43], it can be easily extended to support other languages as well.

5.3.2 *Configuration Parameters*

The second input to the framework is a configuration file for the different building blocks to be used for hardware architecture realization. Through this file, the user can specify: different compute units (e.g. multipliers, adders), process technology to be used (e.g. 16nm, 28nm), the clock frequency of the processor and L1M, and the clock frequency L2M. Moreover, the user can specify the data-width used by the compute units, L1M and L2M. Information to model the L2M burst accesses is also specified in this file: the setup latency for write/read accesses, the type of L2M to be used (e.g. MRAM, SRAM) and the size of the L2M. The different parameters in the configuration file are then used to access a database containing estimates (obtained by synthesis or

2 LLVM used to stand for Low Level Virtual Machine, but this abbreviation has been officially removed.

from specs) of area usage, static and dynamic power, and latency of each of the building blocks. Our L1M implementation uses multiple memory banks of different sizes (see 5.5.1). To estimate the resource usage of the different types of these memories we built a linear model, using synthesis data. We compared the ability of our linear model to predict area, latency and power consumption against the data generated using the synthesis tool and we found it to be accurate - less then 2% error in the area and static energy model and less than 28% error in the dynamic energy model.

5.4 μ-GENIE: ANALYSIS

This section describes the parts of the framework involved in modeling the data transfers between L2M and L1M, *Model L2 read* and *Model L2 write* (5.4.1),the modules that perform data dependence analysis, *Data Dependency Analysis*(5.4.2), and the scheduling of the application operations (5.4.3).

5.4.1 *L2 Memory Read and Write Modeling*

The first operation performed by the framework is to compute the transfer time of the application's input data from L2M to L1M, implemented in the *Model L2 read* block of Figure 5.2. Using static analysis, we obtain details regarding the data structures used in the application. For example, in a matrix vector multiplication kernel, the static analysis extracts information about three data structures: an input matrix and an input vector, containing the input elements of the computation, and an output vector containing the output elements of the computation. An address in L2M is given to each input element used by the application; different data structures are placed in consecutive memory addresses. The entire data transfer is modeled as a single burst-read operation from L2M. The information required to compute the arrival clock cycle of each input element to L1M is extracted from the *Configuration Parameters*. Using this information, the exact clock at which each input element arrives in L1M can be

Symbol	Definition
$AClk_i$	Clock at which element i arrives to L1M
S_r	Setup Latency of a L2M burst read
R_{L2M}	L2M read latency (per read), in L2M clock cycles
Add_{L2Mi}	Offset of element i in the burst access
B_{L1M}	Data bitwidth of L1M
B_{L2M}	Data bitwidth of L2M
Clk_{L1M}	Clock Frequency of L1M
Clk_{L2M}	Clock Frequency of L2M
WBL_{L2M}	L2M write burst latency
S_w	Setup Latency of a L2M burst write
W_{L2M}	L2M write latency (per write) in L2M clock cycles
O	Total number of output elements

Table 5.1: Definition of symbols used in the equations

computed as seen in (1). The equation symbols are described in Table 5.1.

$$AClk_i = S_r + R_{L2M} * (Add_{L2Mi} + 1) * \frac{B_{L1M}}{B_{L2M}} * \frac{Clk_{L1M}}{Clk_{L2M}} \quad (5.1)$$

The schedule produced by the Modified Interval Partitioning (MIP), discussed in 5.4.3, uses the arrival clock cycle computed in this phase to determine when each input element will be available for computation in L1M. The latency of the MIP schedule includes therefore the L2M→L1M transfer, and the computation; it does not take into account the L1M→L2M transfer of the results (phase 3 in Figure 5.3). The *Model L2 write* block in Figure 5.2 computes the latency of the L1M→L2M transfer. The MIP computes the clock cycle at which computation ends (phase 2 in Figure 5.3) and the last data item is written in L1M. The L1M→L2M transfer can start immediately after the last output is generated. The latency of the L1M→L2M transfer is calculated using (2),

$$WBL_{L2M} = S_w + W_{L2M} * O * \frac{B_{L2M}}{B_{L1M}} * \frac{Clk_{L1M}}{Clk_{L2M}} \quad (5.2)$$

where the symbols have been defined in Table 5.1.

5.4.2 *Data Dependency Analysis*

The *Data Dependency Analysis (DDA)* module operates in three stages. The first two stages are the extraction of the *Data Dependency Graph* (DDG)[33] from the application and the schedule of the DDG using the As Soon As Possible (ASAP) and As Late As Possible (ALAP) methodologies. These two steps are core elements in the analysis of high level code for hardware design[32]. Finally, the third step (see 5.4.3) maps DDG instructions to hardware components - or PEs - using a modified *Interval Partitioning* algorithm [53].

To extract the DDG from an application, we use LLVM and custom transformations. We first convert the input application code to its LLVM Intermediate Representation. We then transform the code into static single assignment (SSA) form and perform full-loop unrolling on all of the application loops. After these transformations, there will be no control flow instructions in the application body, and each variable will be defined only once. It is now possible to follow the definition and use chain of the variables to produce a Data Dependency Graph like the one shown in Figure 5.4. The DDG represents each operation as a node - in Figure 5.4 the input and output nodes represent respectively load and store instructions, while oval nodes represent computations - and each edge represents a dependency between operations.

We further process the obtained DDG, aiming to reduce the length of the path between the input nodes and the output nodes. This additional transformation is important because the length of these paths is equivalent to the number of sequential operations required to obtain the outputs, which in turn determines the latency of the application. Taking advantage of operation associativity (where possible) we can transform a long sequence of operations - like the one highlighted in Figure 5.4 - into an equivalent shorter tree.

Next, we apply the ASAP and ALAP scheduling methodologies[32] to the generated DDG. These schedules will associate to each DDG node a clock cycle where the instruction is executed, and *bound* the design space of possible architectures by determining the *maximally parallel* architectures. We start by scheduling the input nodes of the DDG using the arrival clock time of their

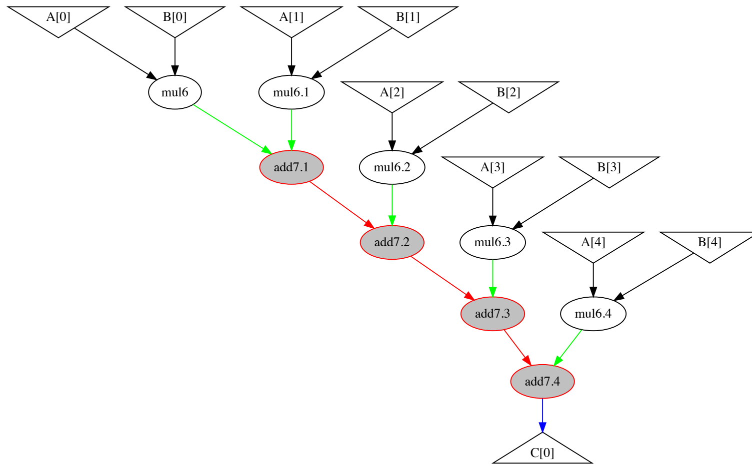


Figure 5.4: A Data Dependency Graph: inverse triangles represent input data, obtained from the *load* instructions; ovals describe operations on data; the triangle at the bottom represents the result, derived from a *store* instruction. Highlighted, a chain of associative operations before being optimized by the *DDA* module (5.4.2).

input data, computed as explained in Section 5.4.1, thus taking into account the L2M - L1M transfer time. Next, we determine the minimal latency required to obtain the outputs of the application with the ASAP schedule: starting from the DDG input leafs, each instruction node is scheduled as soon as its dependencies are resolved. Once ASAP is completed, we can perform the ALAP scheduling: starting from the output leaf nodes, each node is scheduled as late as possible according to its dependencies. Once ALAP is completed, every node is annotated with an ASAP clock cycle and an ALAP one. The difference between these two clock cycles, called instruction *mobility*, identifies an interval in which the instruction can be scheduled without changing the overall latency of the application.

The final stage of the *Data Dependency Analysis* module will allocate the DDG nodes to PEs, leveraging the nodes mobility to minimize the number of PEs of the final hardware architecture.

5.4.3 *PE allocation with Modified Interval Partitioning*

To generate a hardware architecture behaviorally equivalent to the input application and with the latency identified during the ASAP-ALAP scheduling, there are two main requirements: (1) each instruction needs to be computed within its ASAP-ALAP interval, and (2) instructions in the DDG which are executed by the same PE cannot be scheduled at the same time. Our Modified Interval Partitioning (MIP) algorithm - based on the original greedy Interval Partitioning algorithm [53] - is designed to generate, from a DDG, hardware architectures that meet both requirements. Listing 5.1 presents MIP, in pseudo-code. The original Interval Partitioning problem addresses the issue of assigning a number of jobs, with known starting and ending time, to the minimum amount of resources, ensuring that the jobs assigned to a resource do not overlap. To use Interval Partitioning for our problem, we consider instructions as jobs and PEs as resources. There are, however, three main differences between our problem and the canonical Interval Partitioning.

1. The original algorithm considers any job can use any resource, while our architecture requires different PEs for different instructions. We therefore perform interval partitioning several times (lines 5-20), once for each instruction type (e.g., four times for the graph in Figure 5.4). This ensures a correct allocation of instructions to PEs performing the same operation.

2. Due to *mobility*, instructions do not have a fixed starting time. MIP takes the mobility of an instruction into account by allowing a given instruction to start at any time within its allowed interval (lines 11-13).

3. Our instructions are dependent on each other, which is not the case for the jobs in the original interval partitioning. To account for this extra constraint, we ensure that any given instruction (a) is only allocated after its dependencies are allocated (line 4), and (b) is scheduled to start after the ending time of its dependencies (line 7-8).

Listing 5.1: Modified Interval Partitioning (MIP) Algorithm

```
# ASAP[i] and ALAP[i] contain the scheduled cycles for instruction i
# SetPEs is the set of Processing Elements in the architecture
SetPEs=[]
sort instructions by ASAP[i]
for each instruction i
        allocated = False
        dep_deadline = maximum end-time of all instructions depending on i
        schedule[i] = max(ASAP[i], dep_deadline)
        for each PE in SetPEs
                if instruction i matches PE
                        if ALAP[i] >=  next_free_slot[PE]
                                add instruction i to PE
                                schedule[i] = max(schedule[i], next_free_slot[PE])
                                next_free_slot[PE] = schedule[i] + latency(i)
                                allocated = True
        if not allocated
                create new PE with type(i)
                add instruction i to PE
                next_free_slot[PE] = schedule[i] + latency(i)
                add PE to setPEs
```

The MIP algorithm returns `setPEs` and a `schedule` for the current design: `setPEs` is the list of processing elements that form the architecture, with each PE containing the instructions it has to execute, while `schedule` contains the clock cycle at which each instruction is scheduled to be executed.

5.4.4 *Most Parallel and Most Sequential Architectures*

The result of the *DDA* module is the **Most Parallel Architecture (MostPar)**. This architecture takes full advantage of the parallelism of the application and performs the computation with the minimum latency. However, MostPar uses the maximum number of PEs - in the worst case scenario equivalent to the number of instructions in the application - and it will hence have the largest area. At the other end of the spectrum of architectures we can imagine the **Most Sequential Architecture (MostSeq)**, where no parallelism is used and the instruction are scheduled sequentially respecting their dependencies. This architecture will have the worst possible latency, but the minimal impact in area - using only one PE per operation type. Probably none of these two architectures will be of direct interest for the user as they represent two extreme cases. Instead, the interesting architectures are the ones *in between* **MostSeq** and **MostPar**, because they offer interesting trade-offs between power, latency and area. Section 5.5 describes how these intermediate architectures can be generated using **MostPar** and **MostSeq** respectively as upper and lower bounds of the design space.

5.5 μ-GENIE: DESIGN SPACE EXPLORATION (DSE)

The *Design Space Exploration* μ-Genie module generates hardware architectures, behaviorally equivalent to the input application, which exhibit area, latency and power tradeoffs. Our DSE - described in Listing 5.2 - is an iterative process which produces, at the end of each iteration, a different hardware architecture. The iterative process starts its sweep from **MostPar**, and ends when **MostSeq** is generated.

Listing 5.2: Design Space Exploration

```
currentArchitecture = MostPar
found_MostSeq=False
GeneratedArchitectures=[]
while(!found_MostSeq)
    type_count={}
    found_MostSeq=True
    for each PE in currentArchitecture
        type_count[type(PE)]+=1
            if type_count[type(PE)] > 1
                found_MostSeq=False
                break
    if found_MostSeq
       break
    for each instuction i in DDG
        if type(i) == 'store'
            ALAP[i] = ALAP[i]+1
    ALAP=performALAPschedule(instructions ,dependencies)
    SetPEs=MIP(instructions ,ASAP,ALAP)
    GeneratedArchitectures+=[SetPEs]
    currentArchitecture=SetPEs
```

An iteration consists of three steps. First, the instructions corresponding to output leaf nodes in the DDG are selected. The ALAP schedule of these iterations is increased by 1 (lines 14-16), and the ALAP scheduling of the rest of the nodes in the DDG is updated accordingly (line 17). Consequently, the mobility of each instruction node is increased by one. Finally, the MIP is ran again (line 18), using the new ALAP schedule. Due to the increased mobility of each instruction the generated architecture is likely to use less PEs. The process stops as soon as one iteration generates **MostSeq**, which can be recognized because it contains only one PE per operation type (lines 6-13).

There are two side benefits of our DSE approach. First, the user can tune the granularity of the exploration: by increasing the ALAP "slack" beyond 1, the exploration speeds-up, but less architectures are generated. Second, the DSE process can be easily parallelized, because its iterations are independent.

5.5.1 *Architecture Tradeoffs*

For a given input application and a given configuration, μ-Genie outputs a set of hardware architectures - composed of spatial processor and L1M - with different area and latency tradeoffs. Figure 5.5 shows three of the architectures generated during the DSE. Each box represents a PE. The different load and store PEs are implemented as separate L1M banks. As an example, the architecture in Figure 5.5b has 1 L1M bank to store the input data and 5 banks to store the results. This architecture can therefore receive 1 input element per clock cycle from the L2M, and store up to 5 results per clock in the L1M store banks. The remaining PEs are obtained from computing instructions. We allow the architectures to have cycles because the circuit is synchronous, and every instruction has been carefully scheduled. A self-loop in a PE indicates data reuse (see Section 5.6 for implementation details).

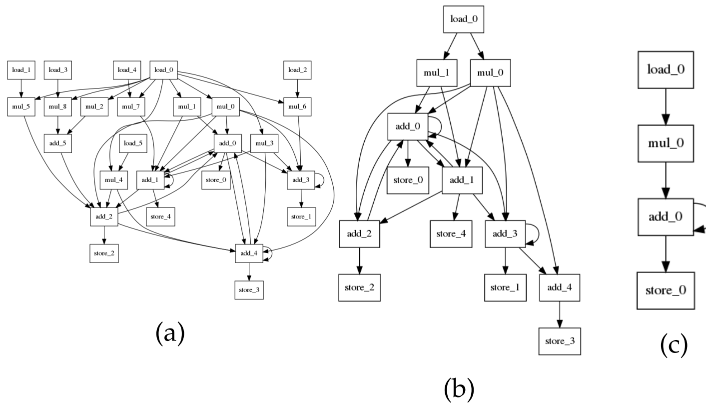


Figure 5.5: Example of architectures generated from a matrix vector multiply application of size 5x5. The MostPar (a), an intermediate architecture (b) and the MostSeq (c).

5.6 ARCHITECTURAL TEMPLATE

To implement in hardware the architectures generated by μ-Genie, we propose a PE template, shown in Figure 5.6. Each PE has an Instruction Memory (IM) where the operations to be performed at each clock cycle are stored. Each instruction is

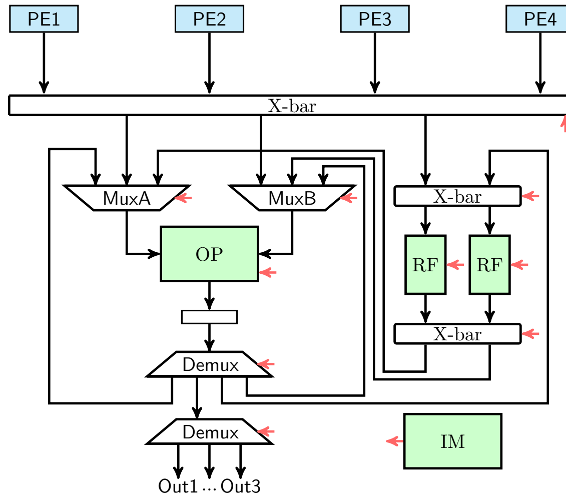


Figure 5.6: Functional Unit template. PE1-PE4 represent "parent" PEs that generate input data. IM is an internal Instruction Memory,where the PE stores the operations to be performed. RFs are internal Register Files, which store reuse data and inputs to be used in the future. OP is the hardware unit actually performing the PE operation.

labeled with the clock cycle in which it should be scheduled. An internal clock counter is compared to the label to decide when to issue the instruction. The internal Register Files (RFs) are used to store input data that needs to be processed in the future, as well as output data that needs to be reused. The white rectangles in the diagram represent configurable crossbars, which can send data from any input port to any output port. OP is the hardware unit that performs an arithmetic (or logical) operation - e.g. add or multiply. This PE template allows modular implementation of the spatial processor architecture - given that the instructions to execute are stored the local IM. The inputs to a PE can either be generated by other PEs or the output generated by the same PE in the previous clock cycle. In addition, the inputs to the PE can be used as operands for immediate computation or stored in the RFs for future use. We have implemented the template PE in RTL that can be configured to build all types of PEs found in the architectures generated by μ-Genie.

5.7 SUMMARY

In this chapter, we presented μ-Genie, a novel framework for codesigning memory-aware custom spatial processors. The framework enables design-space exploration of spatial processor architectures *including* the memory system. We have empirically demonstrated that different spatial processor architectures can be quickly implemented using our novel PE hardware template. In the following chapter, we follow this with the description of the design space explored by μ-Genie, and demonstrate how the framework can be used to perform quantitative analysis of alternative technologies.

6 DSE FOR CODESIGNED COMPUTE AND MEMORY SYSTEMS

The μ-Genie framework presented in chapter 5, allows the procedural generation of spatial architectures where the compute and memory systems are codesigned. Given an application and providing information regarding the technology to be used, we are able to predict latency, area and energy consumption for each of the generated architectures. The selection of one architecture among the set of generated architectures represents a multi-objective optimization problem in an architecture space defined by latency, area and energy consumption.

In this chapter we describe the characteristics of the design space explored by μ-Genie; additionally we describe how the data obtained by our framework can be used to perform quantitative comparisons between different technologies from an application-centric perspective. Specifically, this chapter addresses **RQ4**: Is design exploration a feasible method to codesign parallel-memory computing systems?

This chapter is based on:
Giulio Stramondo, Manil Dev Gomony, Bartek Kozicki, Cees De Laat, and Ana Lucia Varbanescu *"μ-Genie: A Framework for Memory-Aware Custom Processor Architecture Co-Design Exploration"* [74], in *Digital System Design*.

6.1 MULTI-CONFIGURATION DESIGN SPACE EXPLORATION

μ-Genie can also perform design space exploration on the configuration parameters (Section 5.3.2), generating one configuration file for each combination of configuration parameters and aggregating the architectural trade-offs. This allows, as an example, to estimate the effect that different clock frequencies or different memory technology have on the area,latency and energy trade-offs. The use cases discussed in 6.2.2 and 6.2.3 are examples of multi-configuration DSEs.

6.2 CASE STUDIES

In this section, we demonstrate the capabilities of μ-Genie using three case studies. To do so, we analyze the architectures generated by μ-Genie and the energy consumption and latency (or execution time) of each design. We selected two representative applications: matrix-vector (MV) and matrix-matrix (MM) multiplication with matrix sizes 5×5, 10×10, and 15×15. We used the TSMC 28nm target technology library for generating the database containing the area usage and energy consumption of the different building blocks, as required by our framework. We generated multiple input *Configuration Parameters* to let μ-Genie explore the parameter space, and compute the latency, area usage, and energy consumption of the architectures. Each generated architecture has a known latency imposed in each iteration of the DSE (Section 5.5), which we use to compute its static energy consumption. Moreover, after applying the Modified Interval Partitioning algorithm, the instructions performed by each PE are known and this information is used to compute the dynamic energy consumption of an architecture.

In the first case study - Section 6.2.1 - we illustrate how DSE works for 5x5 MV and a single configuration - i.e. one configuration parameter file, see 5.3.2. The second case study - Section 6.2.2 - compares the use of MRAM - modeled according to [23] - and SRAM for L2M (both in 28nm) for the two applications, thus showing the impact of choosing different memory technologies in the L2M. The last case study - Section 6.2.3 - compares μ-Genie architectures for the different MV sizes.

6.2.1 *Single configuration DSE*

The goal of this case-study is to illustrate the ability of μ-Genie to generate, given a single configuration, architectures with different energy consumption. We selected a configuration that uses SRAM in both levels. L2M is clocked at 350MHz, while L1M and the spatial processor are clocked at 1GHz. Figure 7a shows the energy consumption of 30 different pareto-optimal spatial processor architectures, with different latency and energy consumption. We make two observations: (1) the latency and energy consumption

of each design range between the min and max latency, as given by the **MostPar** and **MostSeq** architectures, and (2) as expected, faster designs result in higher energy consumption, due to their larger numbers of PEs.

6.2.2 *MRAM vs SRAM Level 2 Memory*

In this case study we compare the energy efficiency of two alternative technologies to implement L2M: MRAM and SRAM. The comparison is performed using both applications - MV and MM - with 10×10 matrices (see Fig 6.1b and 6.1c, respectively). In both graphs, each point is relative to a hardware architecture generated by μ-Genie; moreover shapes identify different input configurations. Specifically, we compare a total of 18 configurations: 2 L2M technologies, MRAM and SRAM, clocked at 350MHz, and 9 different clock frequencies (400MHz - 1GHz, in steps of 200) for the processor and L1M ensemble.

In both figures we can identify two clusters: LL-HE (low-latency, high-energy) and HL-LE (high-latency, low-energy). Each cluster belongs to one memory technology: LL-HE contains all SRAM designs, while HL-LE contains all MRAM designs. For the MV application (Fig 6.1b) the fastest architecture - using SRAM memory - has a latency of 1375ns and consumes over 1×10^5Joules. The most energy efficient SRAM architecture has instead a latency of 1525ns and consumes under 0.3×10^5Joules, thus being 3x more energy efficient than the fastest, with a latency increase of only 10%. The most energy efficient architecture using MRAM technology has instead a latency of 2210ns and consumes 0.24×10^5Joules, hence having 45% higher latency than the best SRAM counterpart, with 25% lower energy consumption. The matrix multiplication, Figure 6.1c, performs 10 times more operations than the matrix vector multiplication, hence there is a clear overall increase in latency - about 30% - and energy consumption - about 4 times - in comparison to the previous application. In this case the SRAM architecture consuming the least amount of energy has 2% higher latency than the fastest SRAM architecture, but consumes 50% less energy. However, the introduction of MRAM technology in the L2M is not as beneficial as it was for the matrix vector application. The MRAM architec-

ture consuming the least amount of energy has a 3% slowdown compared to the most energy efficient SRAM, while attaining only a 2.2% improvement in energy consumption.

6.2.3 *Different Matrix Dimensions*

In this case study we compare three different matrix sizes for MV - 5×5, 10×10 and 15×15, using the same configurations used in 6.2.2, to evaluate how the energy consumption of an L2M MRAM scales with respect to an L2M SRAM. Figure 6.2 shows the Pareto-optimal architectures generated from each input application. The increase in the matrix size is reflected by an increase in latency: all MV-5×5 application-specific processors have latency below 1000ns, the MV-10×10 have latency between 1250ns and 2500ns, and the latency of all MV-15×15 is beyond 2500ns. However, the increased number of operations results instead in wider trade-offs possibilities. Thus, the normalized latency gap between the most energy efficient SRAM and MRAM, decreases with the matrix size, from 82% for the 5×5, to 42% for the 10x10 and even 32% for the 15×15. The reduction in energy consumption between the same pair of results is instead 20% for the 5×5 and 10×10, while for the 15×15 drops to 16%. Therefore, as the size of the matrix grows, the benefits in energy consumption diminish when using MRAM technology in the L2M. This behavior caused by the increased number of write operations that have high energy impact when the MRAM technology is used.

6.3 RELATED WORK

Previous work on designing spatial processor focuses on the hardware architecture of the processor, while the optimization of the memory system is only partially taken into account. In [59] a spatial processor with distributed control across PE using triggered instructions is presented. Their architecture is built around the guarded-action programming paradigm, where guards - boolean expressions specifying if an action is legal - are evaluated by a scheduler and trigger computations. Support for high level languages is missing, so this spatial processor needs to be pro-

grammed in a low level guarded-action language and the computation needs to be manually mapped on the PEs. Their memory system consist of two levels of memories (L1 and L2) and distributed scratch-pad memories located within the PEs. The design is not tailored for a specific set of applications and they do not perform analysis on the interactions between the memory and processing systems, leaving the modeling of the memory system as future work.

Plasticine, a spatial processor optimized for the acceleration of parallel patterns is presented in [62]. Their memory system is composed of Pattern Memory Units (PMUs) which are connected through a network to Pattern Compute Units (PCUs). Although it allows some degree of configuration, Plasticine is not meant to be optimized around specific applications. The input application needs to be written in a language exposing its parallel patterns - Delite Hardware Definition Language (DHDL) - and then it is mapped on the Plasticine architecture. Hence, the number of memory units (PMUs) and processing units (PCUs), and their interconnections are not optimized around specific applications.

In [3], a framework to generate Application Specific Hardware (ASH) from a C application is presented. The final architecture it produces is asynchronous, and operation dependencies are handled using a token-based mechanism which is implemented in hardware. The memory system of the architecture consists of a monolithic memory. To handle concurrent memory requests the design uses a hierarchy of busses and arbiters, which creates a bottleneck. This means that their memory system is overwhelmed because it is not tailored for the PEs it uses.

Spatially distributed PEs with a dedicated configuration register allow to configure the PEs to one of the operating modes [69] at compile time. Few PEs are connected back-to-back, forming systolic arrays which are then interconnected using an on-chip interconnect. Thus, the processor architecture is quite general-purpose, i.e, the interconnect allows a PE array to be connected to any another PE array. However, there is no automated design flow to efficiently map algorithms to the processor architecture.

An interesting approach is Catena[7], an ultra-low-power spatial processor with a distributed architecture, where multiple techniques - *clock gating*, *power gating* and *voltage boosting* - are

applied in a fine-grained way to optimize energy efficiency. These techniques can be used to explore the power/latency tradeoff of specific applications. However, the impact of the memory system on the performance of the design is not modeled and the memory system is not co-designed with the spatial processor, potentially resulting in an inefficient utilization of the hardware resources; moreover, Catena lacks high-level language support.

In summary, a comparison of μ-Genie against existing work is presented in Table 6.1.

Framework	Type (see 2.3.2)	Application Optimized	Memory Co-Design	Architectural DSE	High Level Language
μ-Genie	Dist. Control	Yes	Yes	Yes	Yes, C
[59]	Dist. Control	No	No	No	No
[62]	Dist. Control	No	No	No	Yes, DHDL
[69]	Dist. Control	No	No	No	No
[3]	Logic Grained	Yes	No	No	Yes, C
[7]	Dist. Control	Yes	No	Yes	No

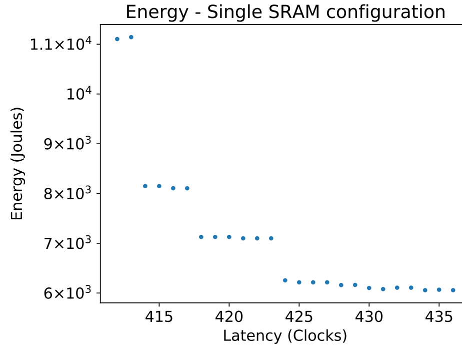
Table 6.1: Comparison with related work.

6.4 SUMMARY

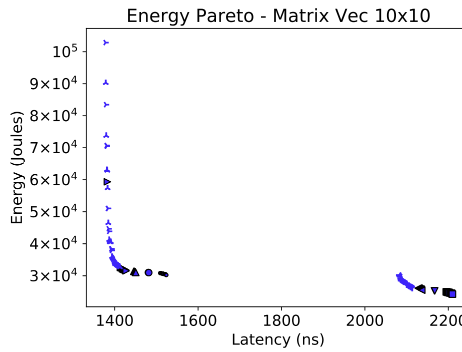
In this chapter we have demonstrated the capabilities of the framework presented in Chapter 5 as a tool for DSE. To this end, we analysed in detail three case studies, which illustrate the sanity of our DSE approach and its ability to facilitate a comparison between the use of MRAM and SRAM technologies. For example, using our approach, we were able to conclude that, for a matrix vector multiplication 10x10, the most energy efficient architecture generated with μ-Genie with MRAM L2M has 45% higher latency than the best SRAM counterpart, with 25% decrease in power consumption.

Future work should focuse on the use μ-Genie to analyze many more applications. A further interesting direction of research is to enhance the framework with the capability of automatically *merging* multiple spatial processor architectures, to generate a single *multi-application spatial processor*. We believe such an ap-

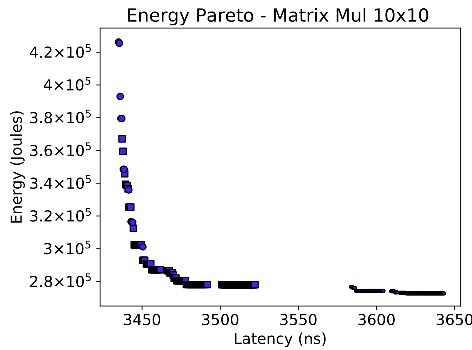
proach would provide new interesting trade-offs, especially in the space of area, latency, and energy efficiency.



(a) 5x5 MM



(b) MV 10x10



(c) MM 10x10

Figure 6.1: Each point represents one μ-Genie spatial processor. Different shapes (in 6.1b and 6.1c) identify different input configurations. 6.1a shows the architecture's Energy over Latency in clock cycles generated from a single configuration of a matrix vector multiplication of size 5×5. Note that (a) presents all designs, while (b) and (c) only include the Pareto-optimal designs.

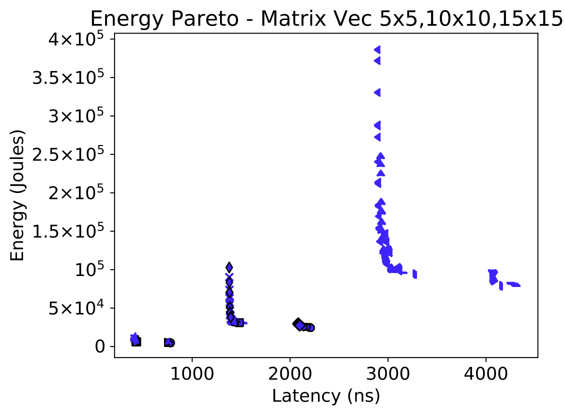


Figure 6.2: Energy Pareto optimal architectures generated by μ-Genie for different sizes of Matrix Vector multiplication 5x5 - with latency ranging from 0 to 1000,10x10 - having latency between 1000ns and 2500ns , and 15x15- with latency above 2500ns. Each point corresponds to an architecture generated by the framework.

7 CONCLUSION

Innovation in the processing system of modern computing systems has seen heterogeneous architectures, combining multi- and many-core processors in complicated processors [5, 70, 80]. However, the performance gap between the memory and computation still increases, and we still see the performance effects of the *memory wall* phenomenon [85]: applications' performance stagnates (i.e., applications hit the memory wall) because accessing data is much slower than processing data. Therefore, rethinking our memory systems becomes a crucial step to enable applications to make efficient use of the processing system. This thesis proposed two novel solutions for the *design and implementation of parallel memory systems*: (1) adapting a polymorphic parallel memory to a given application access pattern, and (2) completely co-designing the memory and processing system to serve a target application. Moreover, these solutions were implemented as prototype frameworks and tools, and validated with relevant case-studies.

7.1 MAIN FINDINGS

In this section, we revisit the main research questions driving this work, and formulate our answers based on our research results.

RQ1*: What is a feasible design for a configurable hardware parallel memory?*

We empirically demonstrated (see Chapter 3) that it is feasible to design a parallel memory system using FPGA technology. Specifically, we presented a configurable parallel memory design (MAX-PolyMem) and performed extensive tests to establish its performance using a Maxeler Vectis DFE FPGA board. The results in Section 3.3 show that our design is able to utilize the entire capacity of the on-chip BRAMs present in the FPGA, implementing parallel memories able to store up to 4MB, having

up to 16-lanes, and supporting up to 4 read ports. The estimated peak performance of MAX-PolyMemis up to 22GB/s for writes and above 32GB/s for reads. Moreover, using the STREAM-Copy benchmark, we were able to confirm that MAX-PolyMem is, in practice, able to attain more than 99% of the estimated peak bandwidth.

RQ2*: Can we define and implement a systematic, application-centric method to configure a hardware parallel memory?*

To answer this research question, we first defined two metrics - *speedup* and *efficiency* - to enable quantitative comparisons between different MAX-PolyMem configurations. We further devised a methodology to systematically determine the optimal (and close to optimal) parallel memory configuration - i.e., the configuration(s) that maximize these metrics - for a given application (see Chapter 4). To showcase the benefits of our methodology, we proposed the Sparse STREAM benchmark, an extension of the STREAM benchmark [79], which includes the original four applications (copy, scale, sum, and triad), but enables benchmark measurements for increasingly sparse datasets. The empirical analysis of our methodology, using the Sparse STREAM benchmark, demonstrated that MAX-PolyMem in combination with our application-centric parallel memory design methodology is not only useful for dense accesses, but also provides significant speed-up for sparser accesses.

RQ3 *- Is there a systematic way to co-design efficient processing and memory systems for a given application?*

Current methods for system design focus on the processing components, and consider the memory system as an add-on. In Chapter 5, we have demonstrated that the co-design of efficient processing *and* memory system is possible with μ-Genie, a framework we devised to systematically generate co-designed spatial architectures. Leveraging the concurrent optimization of processing and memory, our framework is able to minimize the waste of resources by *matching* the performance of the two systems. Moreover, the highly flexible architectural template used

by framework allows for fine-grained tuning of the allocated resources for the target application. Finally, the methodology employed by μ-Genie, is able to generate multiple spatial architectures for a given application, each featuring different latency, area, and energy consumption properties.

RQ4 *- Is design exploration a feasible method to codesign parallel-memory computing systems?*

For our co-designed application-tailored computing systems, we have identified as primary performance metrics **latency, area and energy consumption**. We further showed that, using a multi-configuration DSE, additional dimensions can be added to the design space - e.g., clock frequency and memory technology. We then demonstrated using three case studies that μ-Genie is able to generate designs that provide different latency, area, and energy consumption trade-offs, allowing users to select the architecture that best suits their needs. Finally, we showed that the proposed framework can be used to facilitate comparison between alternative technology, as the SRAM and MRAM memory technologies. For example, we were able to conclude that using MRAM instead of SRAM memories for a matrix vector multiplication 10×10 it is possible to decrease the energy consumption by 25%.

In summary, and we can formulate an answer to the overarching research question of this work:

How can we design and implement efficient application-specific parallel memories?

Our results indicate that, whenever possible, the design an efficient parallel memory system should not be done in isolation, but it should be combined with application optimization. We evaluated two options for this combined approach: application-centric design and codesign. Our application-centric design (Chapters 3, 4) demonstrated performance can be gained from a highly-tuned configuration of a parallel memory, when compared against traditional models. However, it still lead to a memory system with lower efficiency compared to the codesign

approach (Chapters 5, 6). The codesign approach, in fact, guarantees by construction close-to optimal efficiency - and should be preferred.

7.2 MAIN CONTRIBUTIONS

The main contribution this thesis makes are as follows.

- We introduce PolyMem, a Polymorphic Parallel Memory built using BRAMs as a high-throughput software-cache for FPGAs (Chapter 3)
- We perform a DSE analysis to show how MAX-PolyMem scales with the number of lanes, capacity, clock frequency, and peak bandwidth (Chapter 3)
- We provide a systematic approach to optimally configure PolyMem and schedule the set of memory accesses to maximize the performance of the resulting memory system (Chapter 4)
- We define and validate a model that predicts the performance of our parallel-memory system (Chapter 4)
- We present *mu*-Genie, a framework that allows to codesign spatial processor and memory system, explore different architectures automatically generated for a given application, and estimate area, power and latency of each one of the architectures (Chapter 5)
- We propose a sytematic methodology that can be used to perform quantitative comparisons among alternative design choices for codesigned spatial processor and memory system (Chapter 6)

7.3 FUTURE RESEARCH DIRECTIONS

Our work opens up several research directions.

First, PolyMem, the parallel-memory design presented in Chapter 3, can be improved further in terms of flexibility and performance. The work we presented used five mapping schemes (i.e.,

predefined ways to specify the in-memory data layout), and each of these supports different shapes of parallel accesses. A systematic way to explore the space of the possible mapping schemes, and consequently to enable more parallel access shapes would improve the flexibility and performance of the design.

Second, the application-design methodology we use to find a (close to) optimal configuration for MAX-PolyMem, uses static analysis to extract the application access trace. Thus, we only support applications that are statically analyzable. Moreover, during the extraction of the access trace, we assume that the application is completely parallelizable. Both these constraints limit the set of supported applications. To reduce these constraints, and thus extend our approach to more applications, the trace extraction phase may be enhanced using data dependency analysis techniques.

Third, we are actively working on the enhancement of μ-Genie (Chapter 5) with an *architecture merging* module. The spatial architectures currently generated by the framework support the execution of only one application. This merging module allows a user to select two spatial architectures, generated from different input applications, and outputs a combined spatial architecture able to execute both applications. Moreover, it is possible to reduce the resource overhead, letting both application use, whenever possible, the same Processing Elements.

Finally, there is active research in the development of many novel memory technologies (e.g. Magnetoresistive RAM (MRAM) [55], Ferroelectric RAM (FeRAM) [11], resistive RAM [68]). Yet, each novel memory technology has its benefits and drawbacks, and we do not envision any such technology will single-handedly outperform the use of parallel memories. Instead, in the near future, we envision more technologies and parallelism to be used in combination. A hybrid memory system could combine the benefits of different memory technologies, while at the same time using more parallelism to further increase bandwidth and capacity. The μ-Genie framework (Chapter 5), can be easily extended to support further research into such hybrid memory systems.

BIBLIOGRAPHY

[1] Michael Adler, Kermin E. Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. “Leap Scratchpads: Automatic Memory and Cache Management for Reconfigurable Logic.” In: *FPGA '11*. 2011, pp. 25–28.

[2] Debjyoti Bhattacharjee, Farhad Merchant, and Anupam Chattopadhyay. “Enabling in-memory computation of binary BLAS using ReRAM crossbar arrays.” In: *2016 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE. 2016, pp. 1–6.

[3] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. “Spatial computation.” In: *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*. 2004, pp. 14–26.

[4] Paul Budnik and Davis J Kuck. “The organization and use of parallel memories.” In: *IEEE transactions on computers* 100.12 (1971), pp. 1566–1569.

[5] Anastasiia Butko, Florent Bruguier, Abdoulaye Gamatié, Gilles Sassatelli, David Novo, Lionel Torres, and Michel Robert. “Full-system simulation of big. little multicore architecture for performance and energy exploration.” In: *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*. IEEE. 2016, pp. 201–208.

[6] Cadence Design Systems, Inc. “Tensilica Customizable Processors.” In: *https://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable*.

[7] Joao P Cerqueira, Thomas J Repetti, Yu Pu, Shivam Priyadarshi, Martha A Kim, and Mingoo Seok. “Catena: A Near-Threshold, Sub-0.4-mW, 16-Core Programmable Spatial Array Accelerator for the Ultralow-Power Mobile and Embedded Internet of Things.” In: *IEEE Journal of Solid-State Circuits* (2020).

[8] An Chen. "A review of emerging non-volatile memory (NVM) technologies and applications." In: *Solid-State Electronics* 125 (July 2016). DOI: 10.1016/j.sse.2016.07.006.

[9] Y. Chen, T. Yang, J. Emer, and V. Sze. "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices." In: *IEEE JETCAS* (June 2019). DOI: 10.1109/JETCAS.2019.2910232.

[10] Eric S. Chung, James C. Hoe, and Ken Mai. "CoRAM: An In-fabric Memory Architecture for FPGA-based Computing." In: *FPGA'11*. 2011, pp. 97–106.

[11] Yeonbae Chung, Byung-Gil Jeon, and Kang-Deog Suh. "A 3.3-V, 4-Mb nonvolatile ferroelectric RAM with selectively driven double-pulsed plate read/write-back scheme." In: *IEEE Journal of Solid-State Circuits* 35.5 (2000), pp. 697–704.

[12] Alessandro Cilardo and Luca Gallo. "Improving multibank memory access parallelism with lattice-based partitioning." In: *ACM Transactions on Architecture and Code Optimization (TACO)* 11.4 (2015), pp. 1–25.

[13] C. B. Ciobanu, **G. Stramondo**, C. de Laat, and A. L. Varbanescu. "MAX-PolyMem: High-Bandwidth Polymorphic Parallel Memories for DFEs." In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops*. May 2018, pp. 107–114. DOI: 10.1109/IPDPSW.2018.00025.

[14] C. B. Ciobanu, G. Stramondo, Cees de Laat, and Ana Lucia Varbanescu. "MAX-PolyMem: High-Bandwidth Polymorphic Parallel Memories for DFEs." In: *IPDPSW'18 (RAW'18)*. 2018.

[15] C. Ciobanu, X. Martorell, G. K. Kuzmanov, A. Ramirez, and G. N. Gaydadjiev. "Scalability Evaluation of a Polymorphic Register File: a CG Case Study." In: *Proceedings of ARCS*. 2011, pp. 13–25.

[16] Catalin Ciobanu. "Customizable Register Files for Multidimensional SIMD Architectures." PhD thesis. Delft, Netherlands: Delft University of Technology, Mar. 2013. ISBN: 978-94-6186-121-4.

[17] C.B. Ciobanu. "Customizable Register Files for Multidimensional SIMD Architectures." PhD thesis. The Netherlands: Delft University of Technology, 2013.

[18] C.B. Ciobanu, Georgi Gaydadjiev, Christian Pilato, and Donatella Sciuto. "The Case for Polymorphic Registers in Dataflow Computing." In: *International Journal of Parallel Programming* (May 2017).

[19] Codasip Ltd. *Codasip Studio*. 2019. URL: https://www.codasip.com/custom-processor/.

[20] Francis S Collins, Michael Morgan, and Aristides Patrinos. "The Human Genome Project: lessons from large-scale biology." In: *Science* 300.5617 (2003), pp. 286–290.

[21] J. Corbal, Roger Espasa, and Mateo Valero. "MOM: a Matrix SIMD Instruction Set Architecture for Multimedia Applications." In: *Proceedings of the SC99 Conference*. 1999, pp. 1–12.

[22] Miyuru Dayarathna, Yonggang Wen, and Rui Fan. "Data center energy consumption modeling: A survey." In: *IEEE Commun. Surv. Tutor.* 18.1 (2015), pp. 732–794.

[23] Q. Dong, Z. Wang, J. Lim, Y. Zhang, Y. Shih, Y. Chih, J. Chang, D. Blaauw, and D. Sylvester. "A 1Mb 28nm STT-MRAM with 2.8ns read access time at 1.2V VDD using single-cap offset-cancelled sense amplifier and in-situ self-write-termination." In: *2018 IEEE ISSCC*. Feb. 2018. DOI: 10.1109/ISSCC.2018.8310393.

[24] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. "ShiDianNao: Shifting vision processing closer to the sensor." In: *2015 ACM/IEEE 42nd ISCA*. June 2015. DOI: 10.1145/2749469.2750389.

[25] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. "Dark silicon and the end of multicore scaling." In: *2011 38th Annual international symposium on computer architecture (ISCA)*. IEEE. 2011, pp. 365–376.

[26] J.F. Eusse et al. "Pre-architectural performance estimation for ASIP design based on abstract processor models." In: *SAMOS XIV*. July 2014. DOI: 10.1109/SAMOS.2014.6893204.

[27] Gan Fuxi and Wang Yang. *Data storage at the nanoscale: Advances and applications*. Pan Stanford, 2015.

[28] A.D. Santana Gil, J.I. Benavides Benitez, M. Hernandez Calvino, and E. Herruzo Gomez. "Reconfigurable Cache Implemented on an FPGA." In: *ReConFig'10*. 2010.

[29] Chunyang Gou, Georgi Kuzmanov, and Georgi N Gaydadjiev. "SAMS multi-layout memory: providing multiple views of data to boost SIMD performance." In: *ICS*. ACM. 2010, pp. 179–188.

[30] David T Harper. "Block, multistride vector, and FFT accesses in parallel memory systems." In: *IEEE Transactions on Parallel and Distributed Systems* 2.1 (1991), pp. 43–51.

[31] Friedhelm Meyer auf der Heide, Christian Scheideler, and Volker Stemann. "Exploiting storage redundancy to speed up randomized shared memory simulations." In: *Theoretical Computer Science* 162.2 (1996), pp. 245–281.

[32] C-T Hwang, J-H Lee, and Y-C Hsu. "A formal approach to the scheduling problem in high level synthesis." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 10.4 (1991), pp. 464–475.

[33] Sadahiro Isoda, Yoshizumi Kobayashi, and Toru Ishida. "Global compaction of horizontal microprograms based on the generalized data dependency graph." In: *IEEE Trans. Comput.* 10 (1983).

[34] J. Jeddeloh and B. Keeth. "Hybrid memory cube new DRAM architecture increases density and performance." In: *VLSIT 2012*. 2012, pp. 87–88.

[35] Lech Jozwiak et al. "ASAM: Automatic architecture synthesis and application mapping." In: *Microprocessors and Microsystems* 37.8 PARTC (2013). ISSN: 01419331. DOI: 10.1016/j.micpro.2013.08.006.

[36] H. Jun, J. Cho, K. Lee, H. Y. Son, K. Kim, H. Jin, and K. Kim. "HBM (High Bandwidth Memory) DRAM Technology and Architecture." In: *2017 International Memory Workshop (IMW)*. 2017, pp. 1–4.

[37] Richard M Karp. "Reducibility Among Combinatorial Problems." In: *Complexity of computer computations*. Springer, 1972, pp. 85–103.

[38] Kingshuk Karuri, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. "A Generic Design Flow for Application Specific Processor Customization through Instruction-Set Extensions (ISEs)." In: *SAMOS*. Ed. by Koen Bertels, Nikitas Dimopoulos, Cristina Silvano, and Stephan Wong. Springer Berlin Heidelberg, 2009. ISBN: 978-3-642-03138-0.

[39] Donald E. Knuth. "Computer Programming as an Art." In: *Communications of the ACM* 17.12 (1974), pp. 667–673.

[40] M. P. Komalan, C. Tenllado, J. I. G. Pérez, F. T. Fernández, and F. Catthoor. "System level exploration of a STT-MRAM based level 1 data-cache." In: *2015 DATE*. Mar. 2015.

[41] D.J. Kuck and R.A. Stokes. "The Burroughs Scientific Processor (BSP)." In: *IEEE Trans. on Computers* C-31.5 (May 1982), pp. 363–376.

[42] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis. "Multimedia Rectangularly Addressable Memory." In: *IEEE Transactions on Multimedia* (Apr. 2006), pp. 315–322.

[43] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: Palo Alto, California: IEEE Computer Society, 2004. ISBN: 0-7695-2102-9.

[44] S. Lee, J. Jung, and C. Kyung. "Hybrid cache architecture replacing SRAM cache with future memory technology." In: *2012 IEEE ISCAS*. May 2012. DOI: 10.1109/ISCAS.2012.6271803.

[45] Chun-Gi Lyuh and Taewhan Kim. "Memory access scheduling and binding considering energy minimization in multibank memory systems." In: *Proceedings of the 41st annual Design Automation Conference*. 2004, pp. 81–86.

[46] Henry Markram. "The human brain project." In: *Scientific American* 306.6 (2012), pp. 50–55.

[47] *MaxCompiler*. URL: www.maxeler.com/products/software/maxcompiler.

[48] John D McCalpin. "A survey of memory bandwidth and machine balance in current high performance computers." In: *IEEE TCCA Newsletter* 19 (1995), p. 25.

[49] Sally A McKee. "Reflections on the memory wall." In: *Proceedings of the 1st conference on Computing frontiers*. 2004, p. 162.

[50] Paolo Meloni, Sebastiano Pomata, Giuseppe Tuveri, Simone Secchi, Luigi Raffo, and Menno Lindwer. "Enabling Fast ASIP Design Space Exploration: An FPGA-based Runtime Reconfigurable Prototyper." In: *VLSI Des.* (Jan. 2012). ISSN: 1065-514X. DOI: 10.1155/2012/580584.

[51] Vincent Mirian and Paul Chow. "FCache: A System for Cache Coherent Processing on FPGAs." In: *FPGA '12*. 2012, pp. 233–236.

[52] Sparsh Mittal. "A Technique for Efficiently Managing SRAM-NVM Hybrid Cache." In: *CoRR* abs/1311.0170 (2013). arXiv: 1311.0170. URL: http://arxiv.org/abs/1311.0170.

[53] Dave Mount. *Greedy Algorithms for Scheduling*. 2017. URL: http://www.cs.umd.edu/class/fall2017/cmsc451-0101/Lects/lect07-greedy-sched.pdf.

[54] Onur Mutlu. "Memory scaling: A systems architecture perspective." In: *2013 5th IEEE International Memory Workshop*. IEEE. 2013, pp. 21–25.

[55] Peter K Naji, Mark Durlam, Saied Tehrani, John Calder, and Mark F DeHerrera. "A 256 kb 3.0 v 1t1mtj nonvolatile magnetoresistive ram." In: *2001 IEEE International Solid-State Circuits Conference. Digest of Technical Papers. ISSCC (Cat. No. 01CH37177)*. IEEE. 2001, pp. 122–123.

[56] Pradeep Nalabalapu and R. Sass. "Bandwidth management with a reconfigurable data cache." In: *IPDPS 2005*. IEEE, 2005.

[57] Taecheol Oh, Hyunjin Lee, Kiyeon Lee, and Sangyeun Cho. "An analytical model to study optimal area breakdown between cores and caches in a chip multiprocessor." In: *2009 IEEE ISVLSI*. IEEE. 2009, pp. 181–186.

[58] D.K. Panda and K. Hwang. "Reconfigurable Vector Register Windows for Fast Matrix Computation on the Orthogonal Multiprocessor." In: *Proceedings of ASAP*. May 1990, pp. 202–213.

[59] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, et al. "Efficient spatial processing element control via triggered instructions." In: *IEEE Micro* 34.3 (2014), pp. 120–137.

[60] JongSoo Park, Sung-Boem Park, James D. Balfour, David Black-Schaffer, Christos Kozyrakis, and William J. Dally. "Register Pointer Architecture for Efficient Embedded Processors." In: *Proceedings of DATE*. Nice, France, 2007, pp. 600–605.

[61] *PolyMem Code*. github.com/giuliostramondo/RAW2018.

[62] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. "Plasticine: A reconfigurable architecture for parallel patterns." In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2017, pp. 389–402.

[63] Andrew R. Putnam, Dave Bennett, Eric Dellinger, Jeff Mason, and Prasanna Sundararajan. "CHiMPS: A High-level Compilation Flow for Hybrid CPU-FPGA Architectures." In: *FPGA '08*. 2008, pp. 261–261.

[64] Andrew Putnam, Susan Eggers, Dave Bennett, Eric Dellinger, Jeff Mason, Henry Styles, Prasanna Sundararajan, and Ralph Wittig. "Performance and Power of Cache-based Reconfigurable Computing." In: *ISCA '09*. 2009, pp. 395–405.

[65] A. Ramirez, F. Cabarcas, B. Juurlink, M. Alvarez Mesa, F. Sanchez, A. Azevedo, C. Meenderinck, C. Ciobanu, S. Isaza, and G. Gaydadjiev. “The SARC Architecture.” In: *IEEE Micro* 30.5 (2010), pp. 16–29.

[66] Paulo C Santos, Geraldo F Oliveira, Diego G Tomé, Marco AZ Alves, Eduardo C Almeida, and Luigi Carro. “Operand size reconfiguration for big data processing in memory.” In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE. 2017, pp. 710–715.

[67] John Shalf, Sudip Dosanjh, and John Morrison. “Exascale computing technology challenges.” In: *International Conference on High Performance Computing for Computational Science*. Springer. 2010, pp. 1–25.

[68] Shyh-Shyuan Sheu, Kuo-Hsing Cheng, Meng-Fan Chang, Pei-Chia Chiang, Wen-Pin Lin, Heng-Yuan Lee, Pang-Shiu Chen, Yu-Sheng Chen, Tai-Yuan Wu, Frederick T Chen, et al. “Fast-write resistive RAM (RRAM) for embedded applications.” In: *IEEE Design & Test of Computers* 28.1 (2010), pp. 64–71.

[69] S. Smets, T. Goedemé, A. Mittal, and M. Verhelst. “2.2 A 978GOPS/W Flexible Streaming Processor for Real-Time Image Processing Applications in 22nm FDSOI.” In: *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*. Feb. 2019, pp. 44–46.

[70] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. “Knights landing: Second-generation intel xeon phi product.” In: *Ieee micro* 36.2 (2016), pp. 34–46.

[71] G Stramondo, A.L. Varbanescu, and C.B. Ciobanu. “The Case for Custom Parallel Memories: an Application-centric Analysis.” In: *H2RC*. 2016.

[72] **Giulio Stramondo**, Cătălin Bogdan Ciobanu, Cees de Laat, and Ana Lucia Varbanescu. “Designing and building application centric parallel memories.” In: *Concurrency and Computation: Practice and Experience* (). DOI: 10.1002/cpe.5485.

[73] **Giulio Stramondo**, Cătălin Bogdan Ciobanu, Ana Lucia Varbanescu, and Cees de Laat. "Towards application-centric parallel memories." In: *European Conference on Parallel Processing*. Springer. 2018, pp. 481–493. DOI: 10.1007/978-3-030-10549-5_38.

[74] **Giulio Stramondo**, Manil Dev Gomony, Bartek Kozicki, Cees De Laat, and Ana Lucia Varbanescu. "μ-Genie: A Framework for Memory-Aware Custom Processor Architecture Co-Design Exploration." In: *Digital System Design* (2020).

[75] *STREAM-Copy PolyMem MaxJ Code*. URL: https://github.com/giuliostramondo/PolyMemStream.

[76] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen. "A novel architecture of the 3D stacked MRAM L2 cache for CMPs." In: *2009 IEEE HPCA*. Feb. 2009. DOI: 10.1109/HPCA.2009.4798259.

[77] Steven Swanson, Andrew Schwerin, Martha Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, and Susan J Eggers. "The wavescalar architecture." In: *ACM Transactions on Computer Systems (TOCS)* 25.2 (2007), pp. 1–54.

[78] Synopsys Inc. *Synopsys IP designer*. 2019. URL: https://www.synopsys.com/dw/ipdir.php?ds=asip-designer.

[79] *The STREAM benchmark website*. cs.virginia.edu/stream/.

[80] Kuen Hung Tsoi and Wayne Luk. "Axel: a heterogeneous cluster with FPGAs and GPUs." In: *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*. 2010, pp. 115–124.

[81] Vijay V Vazirani. *Approximation algorithms*. Springer Science & Business Media, 2013.

[82] Luis Villa, Michael Zhang, and Krste Asanović. "Dynamic zero compression for cache energy reduction." In: *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. 2000, pp. 214–220.

[83] Yuxin Wang, Peng Li, Peng Zhang, Chen Zhang, and Jason Cong. "Memory partitioning for multidimensional arrays in high-level synthesis." In: *DAC*. ACM. 2013, p. 12.

[84] Samuel Williams, Andrew Waterman, and David Patterson. *Roofline: An insightful visual performance model for floating-point programs and multicore architectures*. Tech. rep. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2009.

[85] Wm A Wulf and Sally A McKee. "Hitting the memory wall: implications of the obvious." In: *ACM SIGARCH computer architecture news* 23.1 (1995), pp. 20–24.

[86] Hsin-Jung Yang, Kermin Fleming, Felix Winterstein, Annie I. Chen, Michael Adler, and Joel Emer. "Automatic Construction of Program-Optimized FPGA Memory Networks." In: *FPGA '17*. 2017, pp. 125–134.

[87] P. Yiannacouras and J. Rose. "A parameterized automatic cache generator for FPGAs." In: *FPT 2003*. 2003.

[88] Shouyi Yin, Zhicong Xie, Chenyue Meng, Leibo Liu, and Shaojun Wei. "Multibank memory optimization for parallel data access in multiple data arrays." In: *Proceedings of ICCAD*. IEEE. 2016, pp. 1–8.

[89] Jintao Zhang, Zhuo Wang, and Naveen Verma. "In-memory computation of a machine-learning classifier in a standard 6T SRAM array." In: *IEEE Journal of Solid-State Circuits* 52.4 (2017), pp. 915–924.

PUBLICATIONS

List of publications between 2016 and 2020:

- **G. Stramondo**, A. L. Varbanescu, and C. B. Ciobanu. "The Case for Custom Parallel Memories: an Application-centric Analysis." In: H2RC. 2016. url: https://h2rc.cse.sc.edu/2016/papers/paper_23.pdf.
- **G. Stramondo**, C. B. Ciobanu, A. L. Varbanescu, and Cees de Laat. "Towards application-centric parallel memories." In: HeteroPar. 2018, pp. 481–493. DOI: 10.1007/978-3-030-10549-5_38.
- **G. Stramondo**, C. B. Ciobanu, C. de Laat, and A. L. Varbanescu. "Designing and building application centric parallel memories." In: CCPE. 2019, DOI: 10.1002/cpe.5485.
- **G. Stramondo**, M. D. Gomony, B. Kozicki, C. De Laat, and A. L. Varbanescu. "μ-Genie: A Framework for Memory-Aware Custom Processor Architecture Co-Design Exploration." In: DSD 2020.
- L. Stornaiuolo, M. Rabozzi, D. Sciuto, M. D. Santambrogio, **G. Stramondo**, C. Ciobanu, and A. L. Varbanescu. "HLS Support for Polymorphic Parallel Memories." In: VLSI-SoC. 2018, pp. 143–148. DOI: 10.1109/VLSI-SoC.2018.8644899.
- C. B. Ciobanu, **G. Stramondo**, C. de Laat, and A. L. Varbanescu. "MAX-PolyMem: High-Bandwidth Polymorphic Parallel Memories for DFEs." In: IPDPSW RAW. 2018, pp. 107–114. DOI: 10.1109/IPDPSW.2018.00025.
- C. B. Ciobanu, **G. Stramondo**, A. L. Varbanescu, A. Brokalakis, A. Nikitakis, L. Di Tucci, M. Rabozzi, L. Stornaiuolo, M. D. Santambrogio, G. Chrysos, and D. Pnevmatikatos "EXTRA: An Open Platform for Reconfigurable Architectures." In: SAMOS. 2018. DOI: 10.1145/3229631.3236092.

- A. Kulkarni, P. Bahrebar, D. Stroobandt, **G. Stramondo**, C. B. Ciobanu, and A. L. Varbanescu. "A NoC-based custom FPGA configuration memory architecture for ultra-fast micro-reconfiguration." In: ICFPT. 2017, pp. 203–206. DOI: 10.1109/FPT.2017.8280141.

- G. Natale, **G. Stramondo**, P. Bressana, R. Cattaneo, D. Sciuto, and M. D. Santambrogio. "A polyhedral model-based framework for dataflow implementation on FPGA devices of Iterative Stencil Loops." In: ICCAD. 2016, pp. 1– 8. DOI: 10.1145/2966986.2966995.

- L. Stornaiuolo, M. Rabozzi, M. D. Santambrogio, D. Sciuto, C. B. Ciobanu, **G. Stramondo**, and A. L. Varbanescu. "Building High-Performance, Easy-to-Use Polymorphic Parallel Memories with HLS." In: VLSI-SoC 2018, pp. 53–78. DOI: 10.1007/978-3-030-23425-6_4.

SOFTWARE AND DATA

This section references the software and data produced in this work:

- **Max-PolyMem Stream**: Configurable parallel memory template linked to a Sparse-STREAM kernel (Ch. 3, Ch. 4) https:github.com/giuliostramondo/PolyMemStream
- **Polymorphic Register File Simulator** (Ch. 3)https://github.com/giuliostramondo/prf-simulator
- **Max-PolyMem Synthesis Data**: Configurable parallel memory template and syntesis results (Ch. 3, Ch. 4) https://github.com/giuliostramondo/prf_maxeler_samos
- **Access Trace Generation and Analysis**: Tools to generate and analyse application access traces (Ch. 4) https://github.com/giuliostramondo/atrace_utils

SUMMARY

Computing drives a lot of developments all around us, and leads to innovation in many fields of science, engineering, and entertainment. As such, the need for computing is increasing at fast pace. This pace has seen the prevalent use of multi- and many-core processors, where parallelism is a sustainable way (for now) to feed our computing needs. We now see single machines reaching multiple TFLOPs in performance, when combining multi-core CPUs and many-core accelerators.

However, a second bottleneck arises in many of these computing systems: the memory. The so-called "memory wall", a term coined in 1994 by Wulf and McKee [85], is a metaphor for the the significant performance limitations that the memory itself poses for computing systems. Simply put, the memory system is often unable to provide enough data to the computing system, creating a bottleneck and limiting the performance of the entire computing system.

One way to go around the memory wall is to redesign the memory system to support more parallelism, and be better suited for the applications running on the computing system. The work presented in this thesis illustrates different ways in which such a novel design can be approached and deployed, as well as the potential performance gains such novel memory systems can provide.

MAIN CONTRIBUTIONS AND FINDINGS

In this work, we present four alternatives to design and/or deploy alternative memory systems.

A parallel software cache [13]

A first challenge we address in this work is the feasibility of designing and implementing parallel memories in FPGAs, such that they can be further reusable for real applications. To this end,

we present a configurable parallel memory design (Polymem) which acts as a software cache and enables parallel memory accesses for different combinations of access patterns. Our design supports multiple lanes, multiple read ports, and concurrent read and write operations. We further present Max-PolyMem, an implementation of PolyMem for the Maxeler reconfigurable platform. We conduct a thorough design space exploration to empirically determine *the best configurations* - having maximal bandwidth - and/or *the performance bounds* for Max-Polymem. For example, we demonstrate that the design with the maximum read bandwidth is a 512KB memory, with 4 read ports, running at 137MHz, and reaching a peak read bandwidth of 32GB/s.

Application-centric parallel memories [72, 73]

To further enable the use of parallel memories in real applications, we propose a comprehensive approach to designing and implementing application-centric parallel memories based on PolyMem. Our approach enables the acceleration of a memory-bound region of an application by (1) analyzing the memory access to extract parallel accesses, (2) configuring PolyMem to deliver maximum speed-up for the detected accesses, and (3) building an actual FPGA-based parallel-memory accelerator for this region, with predictable performance. We show that our performance model accurately predicts the performance of the memory system (below 1% error in most cases).

A method for application-specific compute- and memory-system co-design [74]

PolyMem offers a feasible approach to deploy a semi-flexible parallel memory for a given application. For increased flexibility, we further investigate the feasibility of the computing- and memory-system *co-design*. Thus, we present μ-Genie, an automated framework for co-design-space exploration of custom-processor architecture and memory system, starting from an application description in a high-level programming language. In addition, we propose a spatial processor architecture template that can be configured at design-time for optimal hardware im-

plementation. The architecture template is then used to validate our methodology in a hardware simulation environment.

Design-Space exploration for compute- and memory-system co-design: a collection of case-studies [74]

Finally, to demonstrate the effectiveness of our μ-Genie, we show extensive DSE experiments in which we co-design custom-processor architectures using different memory technologies. For each generated architecture the framework predicts area, latency and energy consumption allowing a comparison of different designs and technologies. As an example of the insight obtained during our DSE, we were able to conclude that changing the memory system technology (i.e. SRAM to MRAM) it is possible to obtain a 25% decrease in power consumption for a matrix vector multiplication application.

SUMMARY AND FUTURE WORK

This work describes two methodologies for the design and use of application-specific parallel memories. The first methodology leverages PolyMem, a configurable parallel memory design with a fixed set of access shapes, which can be tuned, using an application memory trace, to be an efficient software cache for that specific application. The second methodology, implemented in our μ-Genie framework, provides additional flexibility by co-designing the memory system and computing system, using the application data dependency analysis. As future work, we plan to expand Max-Polymem to support an increased variety of applications. Moreover, we aim to extend μ-Genie to allow the generation of multi-application application-specific architectures.

SAMENVATTING

Computing is de drijvende kracht achter vele ontwikkelingen om ons heen, en brengt ons innovatie in velden van onder andere wetenschap, engineering, en entertainment. We zien de vraag dan ook snel stijgen. Deze snelheid heeft gezorgd voor een prevalent gebruik van multi- en many-core processors, waarbij parallel werken (voor nu) een duurzame manier is om te zorgen dat we genoeg rekenkracht ter beschikking hebben. We zien enkele machines meerdere TFLOPS bereiken qua prestatie. Echter, er is een tweede knelpunt in veel systemen: het geheugen. De zogenoemde geheugenmuur, een term bedacht door Wulf en McKee in 1994 [85], is een metafoor voor de significante limitaties op de prestatie, die het geheugen zelf opwerpt voor computing systemen. Eenvoudig gesteld, het werkgeheugen van het systeem is vaak niet in staat om genoeg data beschikbaar te stellen aan het systeem, zodat er daar een knelpunt ontstaat, en de algehele prestatie van het systeem gelimiteerd wordt. Een manier om deze limitaties te verminderen, is door het systeem zo her in te richten dat het meer parallelisme ondersteunt, en beter geschikt is voor de applicaties die draaien op het systeem. Het in deze thesis gepresenteerde werk laat verschillende manieren van herinrichting zien, en geeft weer hoe dezen kunnen worden benaderd, worden geïmplementeerd, en geeft een beeld van de mogelijke winst die zo'n andere aanpak kan opleveren.

BELANGRIJKSTE BIJDRAGEN EN BEVINDINGEN

In dit werk presenteren we vier alternatieven voor ontwerp en / of implementatie van alternatieve geheugensystemen.

Een parallele software cache [13]

Een eerste uitdaging die we bespreken in dit werk, is de haalbaarheid van het ontwerpen en implementeren van parallelle geheugens in FPGAs, zodat ze hergebruikt kunnen worden voor

echte applicaties. Om dit te bewerkstelligen, presenteren we een te configureren parallel geheugen ontwerp (Polymem), die zich gedraagt als een software cache en die toegang tot parallel geheugen voor verschillende combinaties van toegangspatronen toestaat.

Ons design ondersteunt meerdere banen, meerdere leespoorten, en gelijktijdige lees- en schrijfoperaties. Tevens presenteren we Max-Polymem: een implementatie van Polymem voor het herconfigureerbare platform Maxeler. We voeren een volledige ontwerpruimte verkenning (DSE) uit om empirisch te kunnen vaststellen wat de beste configuraties zijn – maximale bandbreedte – en / of de prestatiegrenzen van Max-Polymem. We laten bijvoorbeeld zien dat het design met de maximale lees-bandbreedte 512KB geheugen, met 4 leespoorten, draaiend op 137Mhz, met een piek lees-bandbreedte van 32GB/s, is.

Applicatie-gebaseerd parallel geheugen [72, 73]

Om verder gebruik van parallel geheugen in echte applicaties mogelijk te maken, stellen we een uitgebreide aanpak van het ontwerp en de implementatie van op Polymem gestoeld, parallel geheugen waarbij de toepassing centraal staat, voor. Onze benadering zorgt voor versnelling van een geheugen-begrensde regio van een applicatie door het (1) analyseren van geheugentoegang om de parallelle toegangen eruit te halen, (2) het configureren van PolyMem om maximale versnelling te leveren voor de gevonden toegangen, en (3) het bouwen van een echte FPGA-gebaseerde parallel geheugenversneller voor deze regio, met voorspelbare prestaties. We laten zien dat ons prestatiemodel nauwkeurig de prestaties van het geheugensysteem kan voorspellen (met een foutmarge van onder de 1% in de meeste gevallen).

Een methode voor applicatie-specifieke rekenkracht- en geheugen-systemen co-design [74]

Polymem verschaft ons een haalbare manier om een semi-flexibel parallel geheugen in te zetten voor een gegeven applicatie. Voor meer flexibiliteit onderzoeken we de haalbaarheid van het reken-

en geheugen-systeem co-design. Dus presenteren we μ-Genie, een geautomatiseerd raamwerk voor co-design-ruimte exploratie van maatwerk-processor architectuur en geheugensysteem, beginnend bij een beschrijving van een applicatie in een programmeertaal van hoog niveau. Vervolgens stellen we een spatiele processor architectuur-sjabloon voor, dat geconfigureerd kan worden tijdens de design-fase, voor optimale hardware implementatie. Het architectuur-sjabloon wordt daarna gebruikt om onze methodologie te valideren in een hardware-simulatie omgeving.

Design-Space exploration for compute- and memory-system co-design: a collection of case-studies [74]

Om tenslotte de effectiviteit van onze μ-Genie weer te geven, laten we uitgebreide DSE-experimenten zien, waar we met verschillende geheugentechnieken de maatwerk processor architecturen co-designen. Het raamwerk voorspelt voor elke gegenereerde architectuur de latentie en energieconsumptie, zodat een vergelijking van verschillende designs en technologieën mogelijk is. Een voorbeeld van een verkregen inzicht tijdens de DSE, is dat we kunnen concluderen dat het wijzigen van geheugensysteem technologie (zoals RAM naar MRAM) ervoor kan zorgen dat er een 25% daling van energieconsumptie voor een matrix vectorvermenigvuldiging applicatie kan plaatsvinden.

SAMENVATTING EN TOEKOMSTIG WERK

Dit werk beschrijft twee methodologieën voor ontwerp en gebruik van applicatie-specifieke parallelle geheugens. De eerste methodologie gebruikt Polymem, een configuureerbaar parallel geheugen ontwerp met een vaste set van toegangsvormen, die afgesteld kunnen worden, door middel van een applicatie geheugen trace, om een efficiënte software cache te zijn voor deze specifieke applicatie. De tweede methodologie, geïmplementeerd in ons μ-Genie raamwerk, voorziet in extra flexibiliteit door het co-designen van het geheugensysteem en het rekensysteem, met gebruik van de applicatiedata afhankelijkheidsanalyse. Voor toekomstig werk, zijn we van plan om Max-Polymem uit te breiden, om een grotere variëteit aan applicaties te ondersteunen.

Bovendien zijn we van plan om μ-Genie uit te breiden, om een generatie aan multi-applicatie, applicatie-specifieke architecturen te kunnen genereren.