Imperial College of Science, Technology and Medicine
Department of Computing

Source-to-Source Compilation of Loop Programs for Manycore Processors

Athanasios Konstantinidis

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Computing of Imperial College London and the Diploma of Imperial College, July 2013

Abstract

It is widely accepted today that the end of microprocessor performance growth based on increasing clock speeds and instruction-level parallelism (ILP) demands new ways of exploiting transistor densities. Manycore processors (most commonly known as GPGPUs or simply GPUs) provide a viable solution to this performance scaling bottleneck through large numbers of lightweight compute cores and memory hierarchies that rely primarily on software for their efficient utilization. The widespread proliferation of this class of architectures today is a clear indication that exposing and managing parallelism on a large scale as well as efficiently orchestrating on-chip data movement is becoming an increasingly critical concern for high-performance software development. In such a computing landscape performance portability – the ability to exploit the power of a variety of manycore chips while minimizing the impact on software development and productivity – is perhaps one of the most important and challenging objectives for our research community.

This thesis is about performance portability for manycore processors and how source-to-source compilation can help us achieve it. In particular, we show that for an important set of loop-programs, performance portability is attainable at low cost through compile-time polyhedral analysis and optimization and parametric tiling for run-time performance tuning. In other words, we propose and evaluate a source-to-source compilation path that takes affine loop-programs as input and produces parametrically tiled parallel code amenable to run-time tuning across different manycore platforms and devices – a very useful and powerful property if we seek performance portability because it decouples the compiler from the performance tuning process. The produced code relies on a platform-independent run-time environment, called *Avelas*, that allows us to formulate a robust and portable code generation algorithm. Our experimental evaluation shows that Avelas induces low run-time overhead and even substantial speed-ups for wavefront-parallel programs compared to a state-of-the-art compile-time scheme with no run-time support. We also claim that the low overhead of Avelas is a strong indication that it can also be effective as a general-purpose programming model for manycore processors as we demonstrate for a set of ParBoil benchmarks.

Declaration of originality

This document consists of research work conducted in the Department of Computing at Imperial College London between 2008 and 2013. I declare that the work presented is my own, except where specifically acknowledged in the text.

Copyright declaration

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.

Acknowledgements

First of all I would like to thank my supervisor Paul Kelly. His open-mindedness, patience, deep knowledge and inherent willingness to discuss and criticize new/radical ideas have been crucial for my progress. In addition, his sharp and detailed feedback on my thesis helped me identify the big picture of my work and improve this document to a large extent. I would also like to thank Codeplay Software and EPSRC for supporting this research. In particular, I would like to thank Andrew Richards and Uwe Dolinsky from Codeplay Software and again my supervisor Paul Kelly (recipient of the EPSRC grant) for giving me the unique opportunity to work on this exciting project. A critical part of my research (Chapters 6 and 5) would not have been possible without the support I received from Professor P. (Saday) Sadayappan from Ohio State University and Dan Quinlan from Livermore Labs for who I would like to express my sincere gratitude. I also thank Professor J. (Ram) Ramanujam and Louis-Noël Pouchet for their support and feedback on Chapter 6. Furthermore, I would like to thank Alastair Donaldson and Anton Lokhmotov for their outstanding support during my first years in this project. I would also like to thank Uday Bondhugula and Armin Größlinger for their decisive help in understanding the polyhedral model and identifying prospective research avenues. In addition, I would like to thank my examiners David Thomas and Michael O'Boyle for their valuable feedback which led to a significant improvement of my thesis.

At this point I would like to thank my best friends or people that have influenced me on a personal level. These people are Michael Chiotinis, Vivianna Chiotini, Fanis Gavalas, Yorgos Katsikis, George Koutsouris, Sara Royuela, Tom Mace, Andrea Kowalski and Martin Kong. You might not know it but without you things would have been significantly different.

Last, but not least, I would like to thank my family. In particular my parents Petroula and Spyros, my sister Magda and brother Michael. Nothing would have been possible without you.

Dedication

This thesis is dedicated to my grandfather Captain Michael and to the island of Iraklia.

As you set out for Ithaca,
hope that your journey is a long one,
full of adventure, full of discovery.

Constantine P. Cavafy

Contents

Abstract **i**

Acknowledgements **iii**

1 Introduction **1**

1.1 Thesis Outline 4

2 Background **6**

2.1 Manycore Processors 6

2.2 Source-to-Source Compilation 19

2.3 Static Control Programs 23

2.4 Related Work 24

2.5 Thesis Contributions 29

I Analysis and Optimization **32**

3 The Polyhedral Model **33**

3.1 Introduction 33

3.2 Overview 34

3.3 The Domain 36

3.4 The Schedule 37

3.5 Memory Access Functions 39

3.6 Polyhedral Dependencies 39

3.7 Compilation Flow 42

3.8 Polyhedral Libraries 42

4 Automatic Parallelization: The Pluto Scheduling Algorithm 44

4.1 Pluto Scheduling 45

4.2 Resolving Ambiguous Constraints 49

5 *RosePolly*: Design and Implementation of an Object-Oriented Polyhedral Framework 54

5.1 Polyhedral Compilation in Practice: Related Work and Motivation 55

5.2 Overview: The 3-Layer Interface 57

5.3 Layer-1 – The Compilation Interface 58

5.4 Layer-1 usage example 62

5.5 Layer-2 – The Polyhedral Model Interface 64

5.6 Layer-3 – The Math Interface 64

5.7 Conclusions 66

II Code Generation 67

6 Parametric GPU Code Generation for Static Control Programs 68

6.1 Introduction 69
6.2 Parametric Tiling 73
6.3 GPU Mapping 85
6.4 Local Memory Management 89
6.5 Putting It All Together 99
6.6 Experimental Evaluation 102
6.7 Conclusions 117

7 Beyond Static Control Programs : The *Avelas* Runtime System 118

7.1 Overview 118
7.2 The *Tile Bucket* 120
7.3 The *Thread Bucket* 122
7.4 The *Buffer Bucket* 125
7.5 Experimental Evaluation 127
7.6 Related Work 130
7.7 Conclusions 130

8 Conclusions 131

8.1 Future Research and Development 134

Bibliography 134

List of Tables

3.1 Polyhedral Libraries 43

6.1 Experimental evaluation: GPU characteristics 103

6.2 Experimental evaluation: Tile-size search times 116

7.1 List of ParBoil benchmarks for Avelas evaluation 128

7.2 Avelas experimental evaluation results 128

List of Figures

2.1 Floorplan of the VIA Isaiah chip . . . 8
2.2 CPU scaling graph . . . 9
2.3 High-level structure of heterogeneous systems . . . 10
2.4 Die photo and architecture of NVIDIA Kepler . . . 11
2.5 OpenCL platform model . . . 13
2.6 OpenCL execution and memory model . . . 14
2.7 GPU as OpenCL platform . . . 15
2.8 Source-to-source compilation framework . . . 22

3.1 ADI kernel . . . 35
3.2 UML structure of the polyhedral model. . . . 36
3.3 Simple example of polyhedral representation . . . 37
3.4 Loop skewing in the polyhedral model. . . . 38
3.5 Loop fusion in the polyhedral model. . . . 38
3.6 Array access functions in the polyhedral model . . . 39
3.7 Dependence polyhedron and dependence edges . . . 41
3.8 Basic polyhedral compilation flow . . . 42

3.9 Compilation flow and structure of PoCC 42

4.1 Constraint layout sensitivity for Pluto: Motivating example 50

5.1 UML structure of RosePolly 58
5.2 Vertical view of RosePolly layers 58
5.3 The structure of the RosePolly model-extraction function. 59
5.4 The `FlowGraph` of a simple SCoP 60
5.5 Overloaded `RosePollyBuildModel` method. 61
5.6 Example of elusive transitive dependence 62
5.7 Layer-3 interface structure 66

6.1 Tiling parallelisation 70
6.2 Wavefronts of tiles 70
6.3 Non-rectangular wavefronts 72
6.4 Parametric GPU code generation flow 73
6.5 ADI example showing how the intra-tile execution space is produced. 82
6.6 Jacobi-2d example demonstrating the effect of the proposed intra-tile scheduling algorithm 83
6.7 Final version of $\mathcal{P}_{intra}$ for the Jacobi-2d kernel. 84
6.8 Trade-off between parallelism and locality 90
6.9 Buffer-bucket population mechanism 91
6.10 Local memory buffer definition for Seidel-2D 93
6.11 Template of the produced host-code. 100
6.12 Template of the produced device code. 101

6.13 Performance profiles and relative overhead on GTX280 (no thread-bucket) . . . 109
6.14 Performance profiles on GTX280 (with thread-bucket) 110
6.15 Best performances found after tile-size search on GTX280 110
6.16 Performance profiles and relative overhead on GT540M (no thread-bucket) . . . 111
6.17 Performance profiles on GT540M (with thread-bucket) 112
6.18 Best performances found after tile-size search on GT540M 112
6.19 Performance profiles and relative overhead on GTX580 (no thread-bucket) . . . 113
6.20 Performance profiles on GTX580 (with thread-bucket) 114
6.21 Best performances found after tile-size search on M2070 114
6.22 Performance profiles and relative overhead on K20c (no thread-bucket) 115
6.23 Performance profiles and relative overhead on K20c (with thread-bucket) 116
6.24 Best performances found after tile-size search on K20c 116

7.1 Overview of the *Avelas* execution model. 119
7.2 Structure of a tile bucket consisting of 5 entries each one carrying 3 elements. . . 121
7.3 Storage layout of a tile bucket. 121
7.4 Diagonal reordering with and without tile-buckets. 123
7.5 Structure of a thread bucket object residing in off-chip image memory. 125
7.6 Mri-gridding using tile-buckets 129

Chapter 1

Introduction

For more than two decades, microprocessor performance grew exponentially as computer architects could increase clock frequencies and exploit instruction-level parallelism with virtually no impact on power dissipation and cost. Software could directly benefit from this since no considerable changes (if any) to the source code were required in order to exploit the power of newer and faster processors. This situation has changed significantly over the last 10 years with parallelism becoming increasingly ubiquitous and memory-hierarchy-aware optimizations a critical concern for high-performance software. A milestone in this paradigm shift was the advent of manycore processors primarily represented by *Graphics Processing Units* (GPUs) that have been developed to support massively data-parallel computations. In today's era of *Big Data*[1], manycore processors are becoming vital for a growing number of applications that require extremely fast or real-time processing of huge and complex data-sets. However, programming these devices remains challenging even though the release of *CUDA* and *OpenCL* – two high-level languages for general-purpose manycore programming – has helped manycore processors to become widely adopted as mainstream software accelerators.

OpenCL and CUDA provide a reasonably high-level programming abstraction that enables a wide range of developers to unlock the computing power of manycore chips. Nevertheless, several machine-dependent performance aspects remain exposed to the OpenCL/CUDA paradigm

[1] Big data, is a term denoting data-sets so large and complex that require unconventional processing methods.

which raises the question of *performance portability* – how can we write a program that runs fast accross different manycore chips without the need to change. For example, an OpenCL/CUDA programmer is responsible for managing the granularity of parallelism – the processing of data in chunks of finite size that reflect the underlying finite compute and/or memory resources – as well as the orchestration of data movement from off-chip main memory to on-chip scratchpad memories – a process highly dependent on machine-specific resource trade-offs. Performing these tasks very efficiently for one machine does not mean that the same program will run fast on an other as well. The question of performance portability is particularly relevant today as manycore processors become increasingly pervasive and diverse (from mobile devices to high-performance computing clusters).

For sequential or even vector processors, performance portability is attainable today through any modern retargetable optimizing and/or vectorizing compiler. However, manycore processors – or in fact any parallel processor – pose new and still largely unresolved challenges with respect to performance portability. These challenges could be reduced into the management of two main properties of a program namely parallelism and locality. Unlike scalar or vector optimizations, parallelism and locality management can be effectively captured by high-level source languages like OpenCL and CUDA. Consequently, source-to-source compilation appears to be a highly convenient methodology for studying performance portability as it allows us to concentrate on parallelism and locality management without worrying about the source-to-binary compilation path which is left to existing highly-efficient and trusted optimizing compilers.

In order to avoid the intractability of a generic strategy towards performance portability, i.e. a strategy applicable to any possible program, this thesis concentrates on loop-programs and more specifically to those loop-programs amenable to automatic parallelization based on the polyhedral model (a set of programs also known as *Static Control Programs* or SCoPs). The polyhedral model is a mathematical model of SCoPs that utilizes robust mathematical tools for analysis and optimization. Apart from its intellectual appeal, it is also a mature automatic parallelization technology used by popular commercial compilers like IBM's XL compiler, GCC, LLVM and the R-stream compiler. Even though restrictive, we believe that SCoPs represent a decent subset of loop-programs found in several important high-performance applications and

could also be considered a solid first step towards more generic approaches to performance portability.

In this context, the contributions made by this thesis are the following:

Analysis and Optimization in the Polyhedral Model

Analysis and optimization in the polyhedral model has been an active area of research for more than 20 years (it actually goes back to the modeling and optimization of systolic programs). However, even though important advances have been made in this area, utilizing them in practice is hard as most polyhedral compilers are designed to be used as monolithic executables with restrictive applicability and non-flexible behavior. As a result, we believe that there is a need for a polyhedral framework that leverages existing advances in polyhedral analysis and optimization yet provides a well-defined object-oriented API that is easily extensible/customizable and also allows us to use and manipulate different components of the model intuitively in a programmable fashion. This thesis presents the design and implementation of the first polyhedral framework that meets these requirements.

Code Generation

Even though code generation in the polyhedral model has been successfully addressed for sequential and OpenMP targets, the problem of CUDA or OpenCL code generation remains open. It involves the task of partitioning and mapping a parallelized SCoP into an OpenCL/CUDA execution environment as well as producing additional code for managing the on-chip memory hierarchy. Even though recent efforts towards that direction have shown promising results, this thesis presents a novel code generation algorithm that relies on a well-defined platform-independent run-time environment, called *Avelas*. By using Avelas as our target source language (instead of OpenCL or CUDA) we are able to simplify code generation to a large extent and produce code that is platform-independent and thus can target a variety of manycore chips and platforms – something that has not been attempted by any previous work that we know of.

Performance Tuning

Relying solely on a single compilation step to produce high-performance OpenCL/CUDA code requires compile-time knowledge of the hardware along with a performance model that utilizes that knowledge in order to accurately estimate performance. An alternative is to iteratively optimize a program in search for the best combination of optimization parameters – a process widely known as *auto-tuning*. This thesis follows a third approach that relies on a single compilation step that produces parameterized versions amenable to run-time auto-tuning. Such approach allows us to decouple compilation from the performance tuning process and therefore combine simple and portable code-generation with fast and effective performance tuning.

1.1 Thesis Outline

This thesis is divided into two main parts reflecting the two main concerns involved in source-to-source compilation namely analysis and optimization (Part I) and code generation (Part II). Chapter 2 precedes both parts as it introduced the technical background of the thesis. In particular, the remainder of this document is organized as follows:

Chapter 2 introduces the general context of this thesis including terminology, notation and an extensive coverage of related work. More specifically, it discusses the following four main components of the thesis topic: (i) Manycore Processors (Section 2.1), (ii) Source-to-Source Compilation (Section 2.2), and (iii) Static Control Programs (Section 2.3). The related work section (Section 2.4) is divided into two parts. The first part (Section 2.4.1) presents previous work focusing on source-to-source compilation of SCoPs targeting GPUs. The second part (Section 2.4.2) covers general source-to-source compilation work that aims to raise the level of abstraction for manycore programming. Finally Section 2.5 presents a list of specific technical contributions made by this thesis along with a list of publications produced.

Part I: Analysis and Optimization

This part focuses on the analysis and optimization phase of source-to-source compilation. In our context this phase involves the polyhedral model – an analytical framework for analysis and optimization of SCoPs. The primary objective of this phase is to perform machine-independent automatic parallelization and locality optimization. In particular:

Chapter 3 presents an overview of the *polyhedral model* – the analytical framework that was used for machine-independent automatic parallelization and locality optimization.

Chapter 4 presents the *Pluto* scheduling algorithm – a state-of-the-art algorithm for automatic parallelization in the polyhedral model. Pluto can be used to expose parallelism at the finest granularity and enable the loop-tiling transformation. In this thesis Pluto is used as the main machine-independent loop optimizer. It is shown that in some cases the pluto algorithm can be sensitive to the layout of dependence constraints and a simple solution is proposed.

Chapter 5 presents the *RosePolly* framework for polyhedral compilation. RosePolly is a novel object-oriented API for polyhedral compilation based on the ROSE compiler infrastructure.

Part II: Code Generation

The code generation part focuses on mapping a parallelized SCoP into a GPU execution environment and it is organized as follows:

Chapter 6 presents and evaluates a novel code generation mechanism for SCoPs. The proposed method produces parametrically tiled and platform-independent code amenable to portable run-time exploration of partitioning parameters, i.e. tile-sizes.

Chapter 7 presents and evaluates *Avelas*, a new programming model for manycore processors inspired by Chapter 6.

Finally, Chapter 8 reflects on the contributions made by this thesis and concludes with suggestions for future work.

Chapter 2

Background

This thesis is about source-to-source compilation of static control programs for manycore processors. In this chapter the technical background of this topic is presented after being decomposed into the following counterparts:

- Manycore Processors – Section 2.1
- Source-to-Source Compilation – Section 2.2
- Static Control Programs – Section 2.3

In addition, Section 2.4 presents an extensive coverage of related work to date while Section 2.4.3 presents the motivation behind the chosen research avenue. Finally, Section 2.5 presents a list of specific technical contributions made and a list of resulting publications (Section 2.5.1).

2.1 Manycore Processors

Manycore processors is an emerging class of hardware architectures primarily represented by modern Graphics Processing Units (GPUs) that have been developed to support massively data-parallel[1] computations. In this section we are going to see how microprocessor performance

[1]Data-parallelism – as opposed to task-parallelism – enables the underlying hardware to execute the same operations on multiple data items concurrently.

scaled until the early 2000s and then halted (Section 2.1.1) and why manycore processors provide a viable solution to the performance scaling problem (Section 2.1.2). Finally, we are going to see how manycore processors can be effectively programmed through the OpenCL/CUDA paradigm (Section 2.1.3) and then how their performance potentials can be attained in practice (Section 2.1.4).

2.1.1 Moore's Law and Performance Scaling

Even though the physical properties of the logical building block of modern microprocessors – the semiconductor transistor – were discovered in 1947[2], it wasn't until the late 1950s that we were able to combine them into integrated circuits. Less than 10 years later – in 1965 – Gordon Moore[3] predicted the doubling of the amount of transistors on integrated circuits every year [Moo65], heralding the advent of the silicon revolution [Bon98]. Although Moore's prediction was later adjusted to half the initial rate, the underlying linear trend was soon apparent and became widely known as *Moore's Law*.

A crucial observation pertaining Moore's law is that increasing the transistor count of a microprocessor does not necessarily correlate with performance – the amount of meaningful computation performed per time unit. As a result, scaling performance with the amount of transistors became a critical objective. Initially, the primary avenues towards that goal was to increase the clock frequency on one hand and to exploit instruction-level parallelism (ILP) [HP11] on the other. Even though, power dissipation was an important obstacle to increasing clock speeds at first, the emergence of CMOS[4] technology in the early 1980s along with the scaling properties of MOSFET[5] [DGR+74] enabled us to scale clock frequencies proportionally to the transistor count with practically no impact on power dissipation and cost [FM+11].

As processor performance began to grow throughout the 1980s and 1990s, a widening gap

[2]By John Bardeen, Walter Brattain and William Shockley for which they won the Nobel prize in physics in 1956.

[3]Three years later Gordon Moore co-founded NM electronics which later became Intel Corporation.

[4]Complementary Metal-Oxide-Semiconductor or CMOS, is an integrated-circuit fabrication technology patented by Frank Wanlass in 1967.

[5]Metal-Oxide-Semiconductor Field-Effect Transistor or MOSFET, is the building block of modern CMOS chips. It was invented by Dawon Kahng and Martin M. (John) Atalla at Bell Labs in 1959.

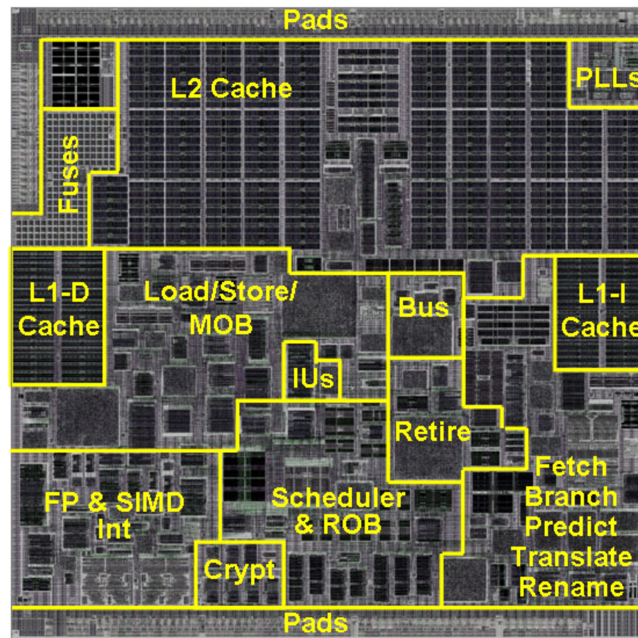


Figure 2.1: Floorplan of the VIA Isaiah chip released in 2008. Most chips had a similar structure in the early 2000s. Notice that computation is carried out by a small portion of the die – namely the Floating Point and SIMD unit – located at the lower left section.

between processor demand for data and memory's ability to deliver, gradually surfaced [WM95]. The solution came from integrating one or more levels of memory hierarchy to the chip as a way of mitigating memory latency by automatically exploiting spatial and temporal locality. These memory modules – known as L1, L2 or L3 caches – were managed by increasingly complex hardware schemes [PH09], that began to occupy a large portion of die[6] space. Figure 2.1 shows how a microprocessor looked in the early 2000s. Note that by that time, the actual compute unit of the chip was covering a small portion of the die area while the rest of the space was covered by mechanisms designed to keep a single serial execution stream active for as long as possible.

Up until the early to mid 2000s, microprocessor performance had sustained an exponential growth that at the same time enabled software to directly benefit without the need to change. Unfortunately, this growth was gradually disrupted primarily due to power dissipation concerns [FM+11] while on the other hand, opportunities for instruction-level parallelism had already reached the point of diminishing returns a few years earlier [HP11, HD04, AHKB00] – around the late 1990s. The graph of Figure 2.2 shows how clock frequencies and ILP began to

[6] A small block of semiconducting material, on which a given functional circuit is fabricated

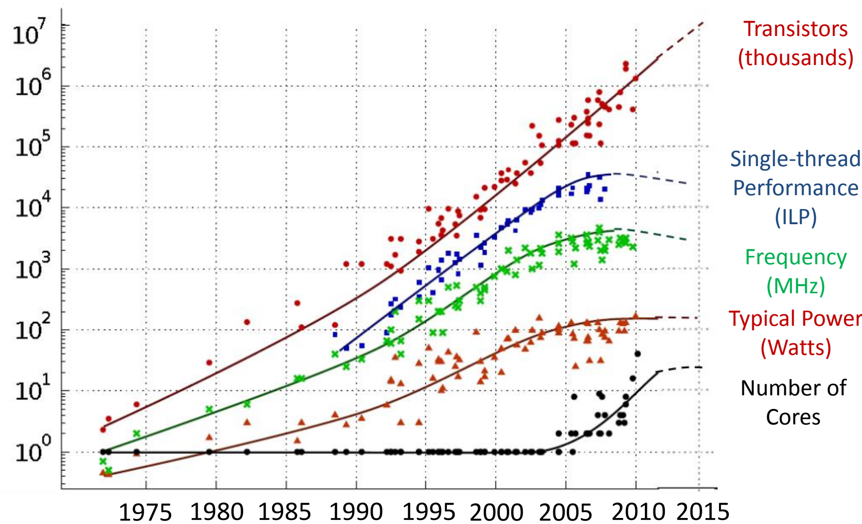


Figure 2.2: CPU scaling graph indicating the end of performance scaling based on frequency and ILP scaling. The Original data were collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten. The dotted line extrapolations were made by C. Moore [Moo11].

plateau (along with energy dissipation capacity) indicating the need for a potentially radical new perspective [ABC+06].

A way around this performance-scaling bottleneck was provided by Chip Multiprocessors (CMPs). In particular, the replication of a single processing core within the same chip could theoretically maintain a linear scaling of performance, e.g., doubling of the transistor count means doubling of processing cores. Such scaling though could only be attained in practice by explicitly exposing and managing coarse-grained parallelism at the software level if such parallelism exists. In other words, performance could no longer maintain the growth rates of the past without considerable software intervention, i.e., programming effort.

Even though exploiting transistor densities with parallelism was proven beneficial before [CSB92, ONH+96], power dissipation constraints remain a limiting factor for CMP scaling today, especially under 90nm[7] fabrication. As a result performance scalability with transistor density remained an open problem in the mid 2000s.

Today, the prevailing solution towards scalable performance comes from a class of hardware configurations commonly known as *heterogeneous systems*. As shown in Figure 2.3 they con-

[7]Under 90-nm fabrication leakage current starts to affect overall chip power as explained by Fuller et. al [FM+11].

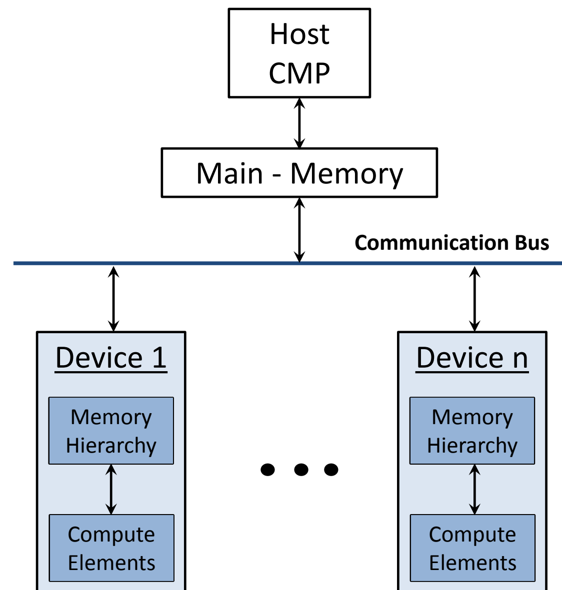
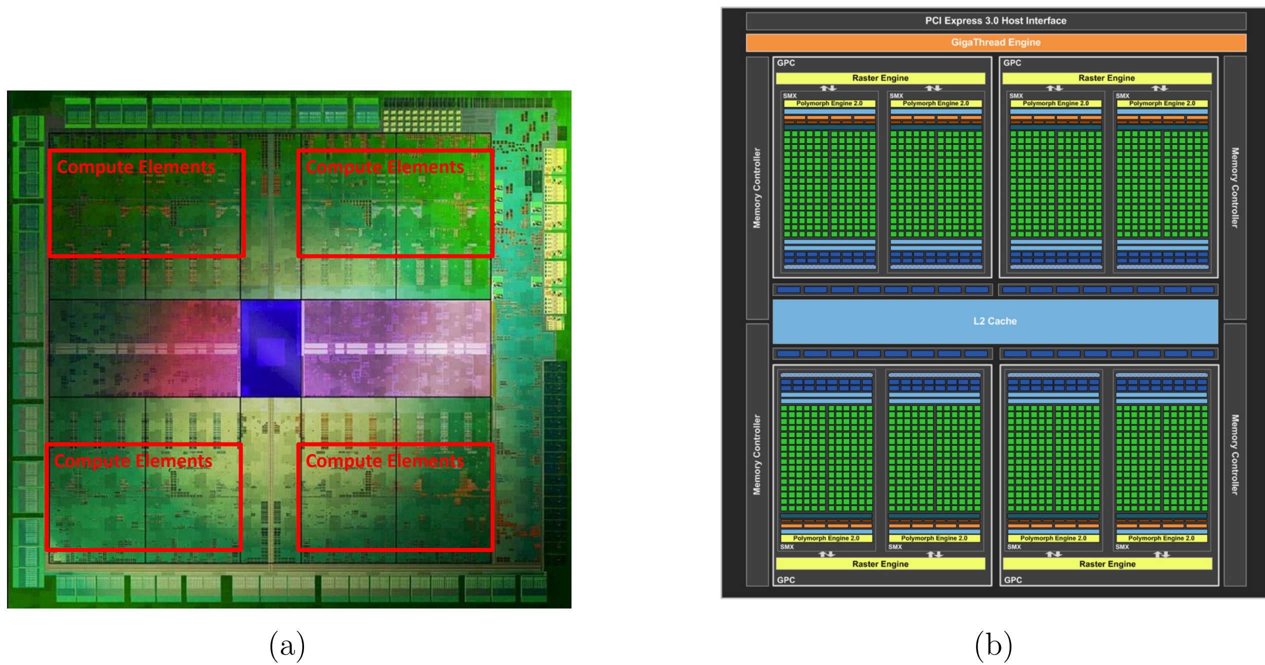


Figure 2.3: High-level view of a heterogeneous system consisted of n devices

sist of a Host CMP running an operating system and coordinating one or more independent devices designed for specialized or general-purpose software acceleration. These devices reside on the same computer node and they could be FPGA cards, Digital-Signal Processors (DSP) or general-purpose Graphics Processors (GPUs). Their main design principle is to limit power dissipation by simplifying the architecture of the chip and rely mostly on software for their efficient utilization.

The emergence of heterogeneous systems today, indicates that the responsibility for performance scaling is being increasingly shifted towards software which would in theory enable performance to scale with transistor densities for the next 5-10 years [FM+11]. What remains to be seen is whether software development for these devices will confirm such scaling projections in practice and what would be the role of new programming environments and compilers in the process. The subject of this thesis is to investigate the ways in which source-to-source compilation can provide an answer to this question for a particular kind of devices namely general-purpose manycore processors or GPUs. More specifically, we will be looking at performance portability – how parallelism and locality can be effectively managed for a variety of GPU devices – for an important subset of loop-programs and how source-to-source compilation can help us achieve it.



(a) (b)

Figure 2.4: (a) Die photo of the NVIDIA gtx680 graphics processor (b) The NVIDIA Kepler architecture implemented by the gtx680 and carrying 1536 parallel compute cores.

2.1.2 Manycore Processors for General-Purpose Computing

The main design principle of heterogeneous devices is to limit power dissipation and attain better scaling properties by reducing the architectural complexity. As such, manycore processors exploit the increasing transistor densities with large numbers of simple compute cores and lightweight on-chip memory hierarchies. Their design originates from graphics processors (GPUs) which in the early 2000s began to support general-purpose compute capabilities. For the rest of this document the term GPU will be used to denote manycore processors in general, as it is more widely recognised and used in the literature. Figure 2.4 depicts a modern general-purpose graphics processor chip along with the respective architectural design. Notice that the die area dedicated to actual computation is now considerably larger compared to Figure 2.1 and consists of several hundreds of parallel computing elements. It is clear that manycore chips are effective in significantly reducing power dissipation while increasing computing power. Nevertheless, whether the available theoretical peak performance can be exploited by the software and to what degree is an open and multidisciplinary problem (e.g. from compilers, to languages and high-performance computing).

In general, manycore chips can be viewed as Single Instruction Multiple Data (SIMD) architectures. However, they are clearly distinguished from conventional SIMD machines as a single instruction can span millions of data elements, each one representing an individual control path or thread that is free to diverge. In order to describe this execution model, NVIDIA coined the term Single Instruction Multiple Threads (SIMT) which is a more accurate characterization.

The first manycore GPUs with general-purpose compute capabilities surfaced in the mid 2000s. However, they could only be programmed with low-level graphics-based languages (i.e. shading languages) which discouraged non-graphics experts from using them. It was not until NVIDIA released CUDA (Compute Unified Device Architecture) in 2006 that general purpose computing with GPUs really started to take off. In addition, the first release of the OpenCL specification two years later was another milestone as it enabled software portability across different heterogeneous devices and platforms.

Today, CUDA and OpenCL can be easily used to exploit the full power of GPU devices. Consequently, by targeting CUDA and/or OpenCL in a source-to-source compilation path we can effectively study the potentials for performance portability by automating the management of parallelism and locality through compile-time or run-time techniques. In fact this is the main objective and focus of this thesis.

2.1.3 The CUDA/OpenCL Paradigm

The striking similarity between the OpenCL and CUDA programming models allow us to use both terms interchangeably. Their primary difference lies in their implementation, i.e. CUDA is defined as a high-level programming language implemented by NVIDIA's proprietary compiler while the OpenCL standard specifies a set of runtime library calls including a call to a JIT (Just-In-Time) compiler that is supposed to produce the device-specific part of the code (the so-called *Device Code* as we will see later on). In order to facilitate the generality of the definitions presented and used in the rest of the thesis, the OpenCL terminology will be primarily adopted.

According to Gaster et al [GHK+11], OpenCL can be decomposed in four models: the platform

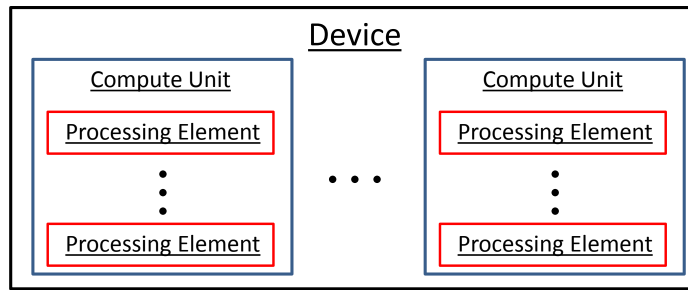


Figure 2.5: OpenCL platform model of a single device.

model, the execution model, the memory model and the programming model. Note that the same models are also evident in CUDA with negligible differences.

The platform model defines a central control processor called *Host*, that coordinates a vector of parallel *Devices* as shown in Figure 2.3. This thesis is focused on a single device configuration i.e. an OpenCL platform consisting of a single GPU device. Regardless of the type (e.g. GPU, FPGA etc.) each OpenCL device is defined by a set of functionally independent *Compute Units* and each Compute unit by a set of *Processing Elements* as illustrated in Figure 2.5. We will refer to this abstract device model as the *physical processor space* of an OpenCL device. The close resemblance between the OpenCL device model and the actual architecture of a GPU is indicative of the importance of GPUs in heterogeneous computing systems.

According to the OpenCL execution model, each device executes a *Device Code* following a data-parallel *SIMT* concurrency model (Single Instruction Multiple Threads). Device code is written as special C-Style functions called *kernels* that are invoked by the Host. The SIMT concurrency model indicates that a single Kernel is executed concurrently across a set of threads called *work-items*. Work-items are organized into *Work-Groups* and work-groups into an *ND-Range* according to the following definitions :

Definition 1. *A Work-Group is a 3D organization of work-items defined by a tuple* $WG(x, y, z)$*, where* x*,* y *and* z *are ranges of work-items.*

Definition 2. *An ND-Range is a 3D organization of homogeneous work-groups defined as a tuple* $NDR(x, y, z)$*, where* x*,* y *and* z *are ranges of work-groups.*

We will refer to this abstract execution model as the *virtual processor space*. Work-items,

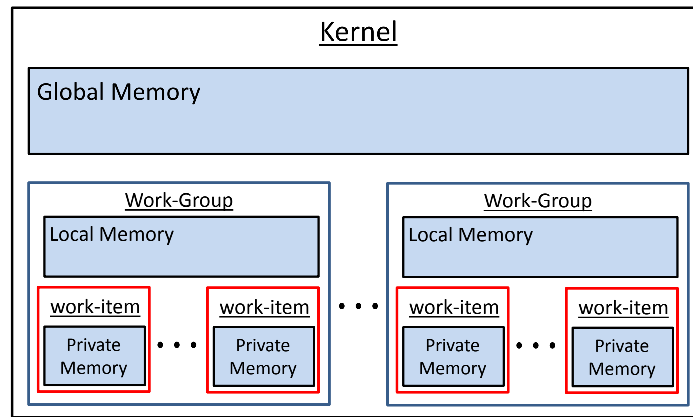


Figure 2.6: OpenCL execution and memory model.

work-groups and nd-ranges can be addressed from the device code through special index variables. We define these variables as :

wi Work-item ID that carries the work-item's position within a Work-Group through a set of coordinates $wi.x$, $wi.y$ and $wi.z$.

wg Work-Group ID that carries the Work-Group position within an ND-Range through a set of coordinates $wg.x$, $wg.y$ and $wg.z$.

glw Work-Group ID that carries the global position of a Work-Group.

gli Work-item ID that carries the work-item's global position within the Work-Group.

Each work-item associated with a specific kernel execution, is exposed to three levels of abstract memory hierarchy. In particular, there is a *Global Memory* randomly accessed by all work-items, a *Local Memory* with a work-group scope and a *Private Memory* dedicated to each work-item separately. Figure 2.6 concisely depicts the OpenCL execution environment consisted of the execution and memory models combined. Note that this is only an abstraction and does not necessarily correspond to physical memory hierarchy levels. The actual physical mapping of the OpenCL abstract memory hierarchy levels for a typical GPU is depicted in Figure 2.7 as part of an OpenCL device.

Evidently, the virtual processor space represents the available parallelism in a computation that might not correspond to the actual physical processor space. This means that multiple

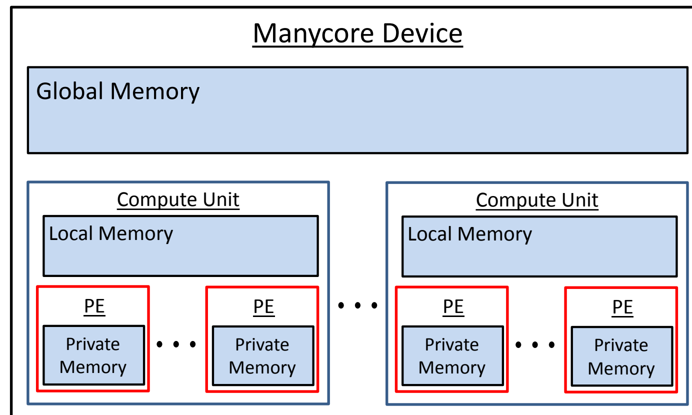


Figure 2.7: A GPU device depicted as an OpenCL platform combined with physical memory hierarchy levels.

Work-Groups might be allocated to a single compute unit at runtime hence share the respective resources. Consequently, carefully weighing the resource usage of each work-group can have a considerable impact on performance as it directly affects the compute unit's capacity to simultaneously execute multiple work-groups. This trade-off is quantitatively captured by the *occupancy* metric defined in paragraph 2.1.4. An interesting demonstration of the importance of this effect is presented by a recent study from Yang et al [YXM+12] that highlights the importance of local memory usage in attaining high GPU performance. In particular, it was shown that if work-groups release their local-memory resources right after they are done with them – instead of holding them for their entire lifetime – then more work-groups will be able to fit in a compute unit yielding much better performance.

2.1.4 Software Performance

The reduced architectural complexity of GPUs signifies the importance of software towards realising their performance potential. This potential is mainly driven by the peak computing throughput of the compute cores on one hand (can exceed 3 TFLOPS today), and the peak main memory bandwidth (can reach up to 208 GB/sec). One of the key requirements in that respect is the ability of software to dispatch massively data-parallel computations or in other words issue instructions that operate on millions of data elements (i.e. threads) in a SIMT fashion. By doing that, GPUs are able to minimize idle cycles by continuously switching between threads

(i.e. work-items) waiting for resources and threads that are ready to be executed.

Having large amounts of data-parallel workloads though is typically not enough. One of the earliest attempts to break down the main components of GPU performance was presented by Ryoo et al [RRB+08] and led to the development of the ParBoil benchmark suite[8]. Soon afterwards, Hong et al. [HK09] and Sim et al. [SDKV12] presented a comprehensive study showing that GPU performance estimation requires detailed static and dynamic profiling information. In particular, an analytical performance model was proposed based on two metrics: Compute Warp Parallelism (CWP) and Memory Warp Parallelism (MWP) that measure the efficiency in exploiting the available compute throughput and main memory bandwidth respectively.

In summary, the main components that influence those metrics (including the ones studied by Ryoo et al.) are the following :

Occupancy The compute elements of a GPU are typically grouped into coarse-grained entities known as *SMs* or *Compute Units* in a CUDA or OpenCL context respectively. These entities contain scarce resources like register files (i.e. private memory) and local memories and can carry up to a maximum number of active work-items i.e. work-items that can be scheduled for execution at any time. Therefore, the resource usage of an individual work-item need to be carefully weighed as it can limit the ability of an SM/Compute Unit to carry active work-items. The ratio between a program's actual active work-items and the maximum amount of work-items possible defines a metric called *occupancy* and is usually directly related to performance. Nevertheless, Volkov [Vol10] showed that better performance can also be achieved at lower occupancy levels by exploiting instruction-level parallelism (i.e. ILP) as we will see later. In principle, occupancy can be calculated statically but a more precise value can be determined through runtime measurements as the number of active work-items can fluctuate unpredictably.

Instruction-Level Parallelism (ILP) As we mentioned in Section 2.1.2, GPUs reduce power dissipation by avoiding complex hardware mechanisms for exploiting ILP automatically

[8] http://impact.crhc.illinois.edu/parboil.aspx

like out-of-order execution and branch-prediction schemes. Consequently, performance is largely exposed to the ordering of instructions and on divergent control flow. The impact of instruction ordering has been experimentally demonstrated by Volkov [Vol10] and incorporated into the analytical performance model of Hong et al [HK09, SDKV12] as well. According to Sim et al [SDKV12], estimating such impact requires analysis of the respective machine code. From a user perspective, minimizing control overhead (i.e. conditional branches) is a good high-level principle for maximizing ILP.

Synchronization On any parallel architecture, communication between parallel processing elements requires some form of synchronization. The large number of active work-items on GPUs indicates that synchronization can be particularly costly in terms of performance and power consumption thus avoiding it if possible can be highly beneficial.

SIMD alignment The total number of work-items on a GPU is divided into equally-sized chunks representing a strictly SIMD execution unit[9]. The importance of the SIMD execution unit lies on the alignment requirements imposed for exploiting spatial locality as well as avoiding serialized work-item diversion in the presence of control-flow. In particular, main-memory data transfers can take advantage of hardware optimizations if certain alignment requirements are satisfied e.g. coalesced accesses [Nvi11, RRB+08]. On the other hand, if work-items diverge within a SIMD execution unit their execution is serialized which reveals the sensitivity of GPU performance to control-intensive code.

Temporal Locality Temporal locality is typically exploited through hardware managed on-chip caches. However, the first general-purpose GPUs relied completely on small software-managed scratchpad memories[10] (of about 16KB) that introduced an additional burden to the software. This burden is partially alleviated on the latest GPUs as they incorporate small non-coherent L1 and L2 caches as well. Even though hardware caching simplifies software development, it also complicates the process of performance modeling as it requires reverse engineering through microbenchmarking [WPSAM10]. On the other hand,

[9] The size of such unit is typically 32. It is called *warp* by NVIDIA and *wavefront* by AMD yet there is no official term in the OpenCL standard.

[10] Note that these memories are reflected by the local memory abstraction of the OpenCL memory model as shown in Figure 2.6.

scratchpad memories are divided into banks that can result in bank conflicts if the same bank is referenced multiple times within a SIMD unit. Such conflicts can serialize scratchpad memory requests thus need to be avoided through padding optimizations [Nvi11].

Special Function Units Expensive math operations like transcendentals and square roots can be efficiently executed by dedicated units. Taking advantage of such capabilities can have a positive impact on performance. However, utilizing special function units is not necessarily a good thing as pointed out by Sim et al [SDKV12], since other compute instructions are required to hide their latency.

Partition Camping Ruetsch and Micikevicius [RM09] demonstrated the importance of partition camping for GPU performance. Partition camping is a phenomenon where multiple main-memory requests are congested within a single main-memory partition. Avoiding partition camping is achieved by rearranging the mapping layout of work-items to data.

Understanding and characterizing GPU software performance has also been studied by Baghsorkhi et al [BDP+10], Bakhoda et al [BYF+09] and Che et al [CBM+09] with the latter work leading to the development of the Rodinia benchmark suite[11].

More specifically, Baghsorkhi et al. [BDP+10] proposes a technique for assisting iterative compilation by providing accurate performance predictions that can be used to narrow-down the tuning space. The proposed method is based on a weighed control-flow graph of a CUDA kernel called *Work-Flow Graph* (WFG), that captures most of the GPU performance aspects we listed here. Even though it is not as detailed as the most recent Sim et al. [SDKV12] approach, it is a robust alternative that uses practical and trusted abstractions like WFG and PDG (i.e. Program Dependence Graph [FOW87]) to model and evaluate CUDA kernels.

Bakhoda et al. [BYF+09] follows a rather different approach by attampting to implement a GPU hardware simulator. As a result, this approach starts from a sophisticated yet well-defined hardware structure and attempts to compute its behavior when executing a GPU program written in NVIDIA's virtual instruction set, i.e. PTX code. The question here is whether

[11] lava.cs.virginia.edu/Rodinia

the assumed hardware structure resembles the actual GPU hardware to a satisfactory degree something that was not sufficiently supported by the paper. Instead, the paper was primarily focused on the effect of specific hardware mechanisms to performance.

Finally, Che et al [CBM+09] follows an application-oriented analysis by proposing a set of GPU benchmarks that stress different combinations of GPU performance bottlenecks.

2.2 Source-to-Source Compilation

A source-to-source compiler is a type of compiler that takes a high-level programming language as input and produces an equivalent program written in another (or the same) high-level programming language. This property is particularly convenient for our purpose because it allows us to distinguish between high-level source transformations like loop transformations (e.g. loop interchange, loop fusion, loop unrolling, loop tiling etc.) and automatic parallelization from low-level concerns like scalar optimizations, instruction scheduling, register allocation etc. The former are the kinds of high-level source transformations we can use for parallelism and locality management while for the later low-level operations we can take advantage of existing highly efficient vendor-specific source-to-binary compilers. For example in our context an NVIDIA CUDA source-to-binary compiler or a vendor-specific OpenCL runtime can be used to produce the final executable program.

Figure 2.8 depicts the structure of a source-to-source compilation system that can be characterized by three main components: (i) the front-end abstraction, (ii) the source-to-source compiler infrastructure and (iii) the back-end abstraction. With respect to the front-end abstraction we see that the more high-level it becomes the more restrictive it gets with respect to applicability. For example a source-to-source compilation scheme that takes a very high-level source language as input – like a domain-specific language – would be restrictive to a certain class of computations that can be expressed with such language. Static control programs are considered a relatively high-level front-end abstraction as they are statically-analyzable loop-programs and thus a restrictive class of programs. However, we believe that SCoPs represent an important

set of loop-programs that can help us understand performance portability better and how to proceed to more generic approaches.

With respect to the back-end abstraction, the more it relies on a run-time system the more simple and portable our code generation algorithm becomes. The price we pay here is the overhead induced by the run-time. This thesis proposes such a run-time system, called Avelas (Chapters 6 and 7), that enables us to formulate a robust and portable code generation algorithm with very low overhead. In fact we show that in certain situations, i.e., execution of wavefront parallelism, Avelas yields substantial speed-ups over a state-of-the-art compile-time method with no run-time support.

Finally, a source-to-source compiler infrastructure is characterized by its *Intermediate Representation* or *IR*. This IR is the main abstraction vehicle for analyzing the input source and carrying the compile-time transformations and optimizations we want to apply to it. We will now look at some of the most popular compiler infrastructures today along with their IR.

LLVM

Started as a research project at the University of Illinois at Urbana-Champaign [LA04], LLVM has now become one of the most widely used open-source compiler infrastructure in the academia and industry as well[12]. It utilizes a Static Single Assignment (SSA) intermediate representation that comes with an extensible/customizable set of optimizations. Today, LLVM consists of a production-quality front-end, called *clang*, that supports C/C++ and Objective-C/C++ and a re-targetable back-end that generates machine-code and can also be used in a JIT fashion. In case of high-level source-based transformations – the ones that we are particularly interested in for reasons explained earlier – LLVM's front-end infrastructure, i.e., clang, provides a set of powerfull utilities for syntax-tree manipulation that are part of *libclang*.

[12] `llvm.org`

ROSE

Unlike LLVM, ROSE [Qui00] was designed to be a high-level source-based transformation/optimization infrastructure similar to the most recent libclang. As a result, it uses an object-oriented IR, called SAGE III, that facilitates high-level manipulation of the program's syntax tree. It was developed in Lawrence Livermore National Labs[13] and today includes a large set of optimization and analysis tools. ROSE was the compiler infrastructure chosen to be the foundation of our research. In fact, the polyhedral framework presented in Chapter 5 has been developed for ROSE and is now part of the ROSE distribution.

SUIF

The SUIF[14] compiler [WFW+94] infrastructure is one of the earliest attempts to realise an extensible compiler framework that facilitates diverse research on compiler analysis and optimization. It is based on a combination of low-level and high-level IR primitives that include loop, conditional statement and array access objects for automatic loop transformations, as well as low-level instructions for robust back-end optimizations and code generation.

PoCC

The Polyhedral Compiler Collection or PoCC [PBB] is a research source-to-source compiler based on the polyhedral model – a robust mathematical intermediate representation of loop-programs. It can actually be viewed as a glue that brings together multiple independent projects in polyhedral compilation through a common communication mechanism embodied by the *OpenScop* intermediate form. It is implemented in C and therefore does not take advantage of object-oriented programming. In Chapter 5 we propose an alternative polyhedral compiler infrastructure that utilizes an object-oriented design for defining an intuitive API as opposed to PoCC's monolithic executable approach.

[13] rosecompiler.org
[14] suif.stanford.edu

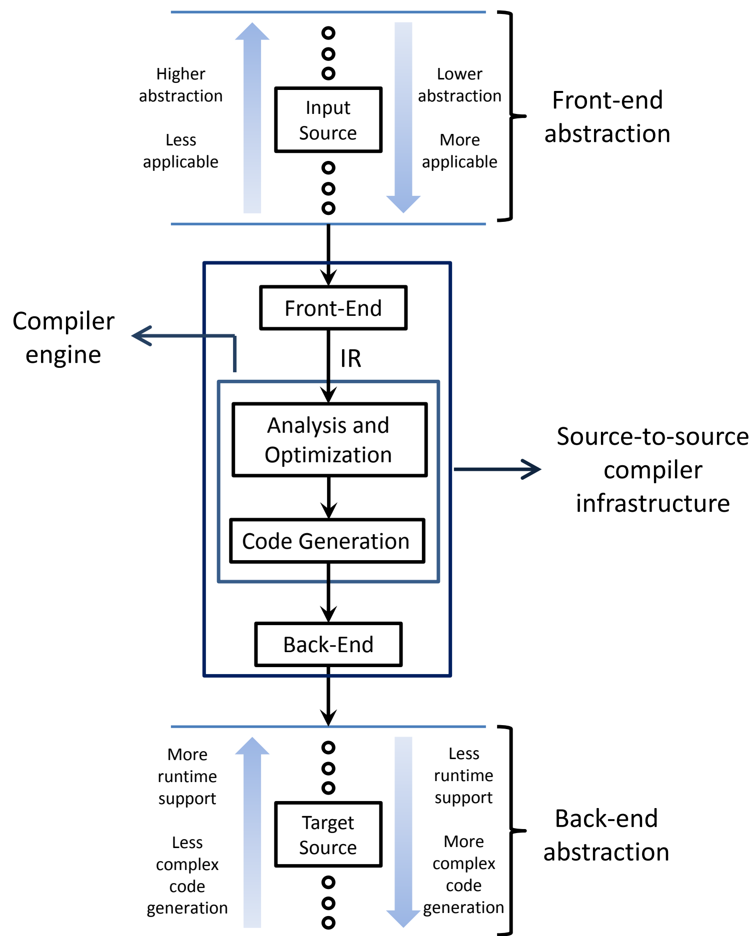


Figure 2.8: Source-to-source compilation framework

Mercurium

Mercurium [BDG+04] is an academic framework developed at the barcelona supercomputing center that currently supports C and C++. It is primarily used to support the *nanos* environment that implements OpenMP. The distinguishing characteristic of Mercurium is that its IR is hidden behind a high-level transformation environment that operates directly at the source through pragma declarations.

Cetus

Cetus [LJE04] is a research compiler framework developed at Purdue University. It is written in Java, and it's IR reflects the ANSI C syntax which in turn limits the extensibility of Cetus to additional languages.

2.3 Static Control Programs

Static Control Programs or *SCoPs*, are loop-based computations with statically determinable control flow and data-access patterns. In other words, their behavior can be analyzed precisely during compilation. Such programs are also called affine [ALSU07] as they meet the following conditions :

Control Flow The only control structures allowed are for-loops and conditional branches. Both must depend on affine expressions of outer loop indices and global symbolic parameters (i.e. problem sizes).

Data Accesses Array access functions must be affine expressions of outer loop indices and global symbolic parameters. Scalar data are treated as 0-dimensional array accesses.

Static control programs, enable us to utilize the polyhedral model [Fea92b, Fea92b, ALSU07] for robust analysis and optimization based on well-known mathematical tools like integer linear programming solvers. They represent computation kernels with inherent data-parallelism that constitute building blocks of several high performance computing applications. They typically consist of a few computation statements associated with multidimensional loop-based execution spaces. They can be exemplified by the polybench suit[15] – the most well-known benchmark suite of polyhedral programs.

The motivation for considering SCoPs as the class of programs we used to investigate performance portability is that they allow us to evaluate a state-of-the-art technology for compile-time automatic parallelization and locality optimization, i.e. the polyhedral model. We believe that such strategy will help us understand the limits of compile-time analysis when it comes to automatic parallelization and locality optimization since the polyhedral model offers a powerfull analytical model of loop-programs – one of the dominant sources of data-parallelism in imperative sequential programs.

[15] http://www.cse.ohio-state.edu/~pouchet/software/polybench

2.4 Related Work

Source-to-source compilation for GPUs represents a large body of academic and industrial research that this section attempts to cover. The work that is most relevant to this thesis will be discussed in Section 2.4.1 and involves research on source-to-source compilation of SCoPs targeting GPUs. Section 2.4.2 will be focusing on general source-to-source compilation work aiming to provide some degree of performance portability and thus reduce programming effort through higher abstraction layers. Finally, Section 2.4.3 presents the relation of this thesis to previous work and highlights the motivation behind the research avenue followed.

2.4.1 From Static Control Programs to OpenCL/CUDA

Baskaran et al [BBK+08a, BRS10] presented the first attempt to realise a C-to-CUDA compiler for SCoPs based on the Pluto scheduling algorithm [BR07, BBK+08b]. Similar frameworks later followed originating from both industrial [LVM+10, ACE+12] and academic research [VCJC+13, GCK+13]. More specifically, Verdoolaege et al. [VCJC+13] presented and evaluated the first robust C-to-CUDA compiler based on the polyhedral model (*PPCG*) that was shown to produce effective CUDA code for the entire polybench suite[16] unlike any previous work. An experimental evaluation showed that PPCG produces similar or slightly faster code than existing polyhedral C-to-CUDA compilers [BRS10, ACE+12].

One of the performance bottlenecks associated with polyhedral C-to-CUDA compilation has to do with wavefront parallelism or *DOACROSS*[17] loops. This issue was studied by Peng and Jingling [DX11] in the context of a model-driven tile-size selection algorithm. In addition, Grosser et al. [GCK+13] proposed a split-tiling mechanism that avoids wavefront parallelism for a certain class of stencil computations.

Armin Größlinger [Grö09] showed that existing C-to-CUDA approaches do not utilize local memory efficiently in cases where an outer sequential time loop executed by each CUDA thread

[16] http://www.cs.ucla.edu/~pouchet/software/polybench

[17] This term is used in the literature to denote wavefront-parallel iteration spaces.

traverses the data elements of a local memory buffer. It was shown that a compile-time algorithm based on polyhedral analysis tools can offer precise utilization of local memory by evicting the data elements that are no longer needed by subsequent time steps. However, the effectiveness of this method was only demonstrated for 1-dimensional problems. For higher dimensional problems the cost of evicting data from local memory might be too high.

Yang et al [YXKZ10] presented an alternative source-to-source compilation scheme that was not based on the polyhedral model even though the benchmarks used are SCoPs. In particular, the input source is an already parallelized computation written as a naive CUDA/OpenCL kernel that the compiler attempts to partition and optimize. The effectiveness of this method was demonstrated for linear algebra kernels and compared against the respective CUBLAS[18] methods. Even though this method offers superior performance compared to polyhedral approaches its weakness is the restrictive applicability since it can only be applied to a relatively small subset of SCoPs, i.e. linear algebra computations.

It is important to highlight that all the approaches mentioned in this section rely entirely on the compiler to directly produce CUDA or OpenCL code. Consequently, exploring the resultant design space requires an empirical or model-driven iterative compilation search as indicated by Baskaran et al [BBK+08a] and Yang et al [YXKZ10]. Such approach also leads to a complex, platform-dependent and hard to maintain code generation algorithm. In this thesis (Chapter 6) we propose an alternative source-to-source compilation path that relies on a platform-independent run-time system which enables us to formulate a simple and portable code generation method. Furthermore, the produced code is parametrically tiled which allows us to avoid the cost of iterative compilation induced by all other methods.

2.4.2 Raising the Level of Abstraction

Raising the level of abstraction to ease the burden of parallel and memory hierarchy aware programming has been an active area of research for many years. The emergence of manycore processors introduced challenges that motivated new abstractions and the adaptation of existing

[18] `https://developer.nvidia.com/cublas`

ones. With this survey we wish to identify alternative avenues towards performance portability through source-to-source compilation and how our work relates to them.

One of the first attempts to raise the level of abstraction for GPU programming was made by Ueng et al [ULBH08] with the introduction of *CUDA-lite*. The main focus of CUDA-lite is the memory hierarchy of CUDA devices and how source-to-source compilation can be used to automate their efficient utilization. In CUDA-lite, a set of declarative annotations guide a compile-time analysis engine that identifies global memory access function and ensures their proper alignment. They do that by promoting unaligned accesses to local memory in an aligned fashion. The alignment requirement is then lifted when data reside in local memory. A very similar idea was also pursued by Yang et al [YXKZ10] but with a more powerful analysis that doesn't depend on declarative annotations.

The *hi-CUDA* model proposed by David Han and Tarek Abdelrahman [HA09] uses a set of declarative annotations to sequential code that guide a source-to-source compiler towards generating CUDA code. This concept is similar to CUDA-lite but with a less intelligent optimization engine. In particular, hi-CUDA is not supposed to carry out any sophisticated optimizations but simply to follow the instructions expressed by the annotations. In other words, it can be considered as a simple and platform-independent alternative to CUDA.

A similar objective was pursued by Garland et al [GKZ12]. In particular, an efficiency-oriented (as opposed to productivity-oriented) notation is proposed that facilitates a unified environment across different heterogeneous devices. It is stated that such approach can be used for realizing high-level abstractions or libraries similar to Thrust [BH11] – a principle that is very similar to the motivation behind Avelas (Chapter 7) in the sense that it investigates a common platform-independent layer that can be viewed as a back-end abstraction for source-to-source compilation.

Recently, Dubach et al [DCR+12] presented a compiler technology to support the Java-based Lime language [ABCR10] for heterogeneous computing. The strength of the proposed system is based on the properties of the underlying type system that enables the source-to-source compiler to perform efficient management of the memory hierarchy without resorting to sophisticated alias or dependence analysis.

Recently as well, Bauer et al. [BTSA12] introduced a new programming model for heterogeneous computing called *Legion*. Legion utilizes a runtime system that allocates and manages recursive tasks each one associated with a *logical region* specifying a variety of data-access information (e.g. data-access regions, privileges and coherence information). The effectiveness of Legion was demonstrated for three applications executed on three multi-GPU clusters. In particular, it was shown that Legion programs can achieve software and performance portability on scaling multi-GPU clusters. However, the issue of single-GPU performance portability was not examined.

Along with the introduction of new abstractions for GPU programming, there has been a considerable body of research focusing on adapting existing programming models to GPUs. For example, Hong et al [HCC+10] investigated the feasibility of a *mapReduce* [DG08] implementation for GPU targets by presenting a unified API, called *mapCG*, that serves as a source-to-source back-end abstraction for mapReduce programs. Their work improves over previous mapReduce implementations like *Mars* [HFL+08] and *Phoenix* [RRP+07] by reducing the memory requirements of mapReduce and proposing ways of automating partitioning and scheduling of execution between a CPU and a GPU. Similar work was also done by Chen et al [CHA12].

Adapting OpenMP to support GPU targets has been comprehensively studied by Lee et al [LE10, LME09] and shown to be a promising perspective. It was shown that direct translation of existing OpenMP applications does not always yield good performance and as a result new optimizations are needed especially for ensuring the proper SIMD alignment of global memory accesses. Such optimizations were based on syntactic pattern-matching and loop transformations like parallel loop-swap and matrix transpose for regular applications, and loop collapsing for irregular ones.

Klöckner et al [KPL+09] proposed *pyCUDA* – a high-level runtime system based on the python scripting language that targets CUDA devices. This approach though focuses on runtime code-generation mechanics without attempting to automate any analysis or optimization techniques. Similar work has been demonstrated for Matlab by Fatica and Jeong [FJ07].

The feasibility of using a graph-based intermediate representation for efficient partitioning and

scheduling of execution between a host CPU and a CUDA device, was investigated by Grossman et al [GSBS11] by adapting Intel's Concurrent Collections (CnC) programming model. More specifically two extensions to CnC are proposed namely multithreaded steps and automatic generation of data and control flow between CPU steps and GPU steps.

The effectiveness of directive-based languages has received considerable attention and resulted in highly promising systems. A leading representative is the *OpenAcc* standard examined by Wienke et al [WSTaM12] and Lee et al [LV12]. In fact Lee and Vetter [LV12] present an evaluation of commercial and academic directive-based languages including PGI[19], hmpp [DBB07], OpenMPC [LE10], hi-CUDA [HA09] and R-stream [LVM+10]. Their study though did not include Mint [UCB11], a specialized directive-based language for stencil computations that was shown to be effective for simple stencil kernels.

This section concludes with a reference to domain-specific languages (DSLs). DSLs are specialized languages designed for certain kinds of computations and therefore leverage domain-specific assumptions to assist analysis and optimization [VDKV00]. Datta et al. [DMV+08] presented one of the first attempts to realise a DSL for stencil computations targeting multiple platforms including GPUs. It was shown that iterative compilation combined with a DSL specification can yield performance portability across a variety of hardware. Holewinsky et al. [HPS12] presented a DSL framework for simple stencils targeting OpenCL/CUDA and performed tuning experiments for tile-size selection. The proposed method was based on an overlapped tiling scheme introduced by Krishnamoorthy et al. [KBB+07] and also used by Meng et al. [MS09] in the context of GPUs. Other projects on DSLs for manycore processors include *OP2* [MGR+12, BBL+12], *Nikola* [MM10], *Obisidian* [SCS10], *FLAME* [GGHVDG01] and the work by Cartey et al. [CLdM12].

2.4.3 Conclusions and Motivation

An important observation from the literature survey of Sections 2.4.1 and 2.4.2 is that most previous work relies solely on the compiler to produce the right code for a given GPU device.

[19] http://www.pgroup.com/resources/accel.htm

In that case, performance portability can be achieved either through iterative compilation, or through a single sophisticated compilation step. Perhaps the only exception is the work from Holewinsky et al [HPS12] that proposes a method for run-time tile-size selection using OpenCL as the target language thus aiming a variety of GPU devices. The main weakness of this approach though is the very restricted applicability since it can only be applied on simple stencil programs (e.g. Jacobi method or similar).

Using a platform-independent run-time layer – that exists between a high-level programming abstraction and low-level GPU code – that acts as the target language for a source-to-source compiler has been proposed in the past [GKZ12, HCC+10, HPR+08]. The main motivation here is to simplify the code generation pass of source-to-source compilation and also make the source-to-source compiler easily portable across targets. However, to the best of our knowledge, none of the proposed run-times are used for run-time performance tuning. In other words, existing run-time systems are not designed to handle parametric input thus rely on the compiler to provide the exact execution parameters (e.g. partitioning parameters like work-group sizes and local memory usage) or even make those decisions automatically.

In this thesis we present a novel source-to-source compilation path that generates parametrically tiled GPU code for an important set of loop-programs, namely SCoPs. Such source-to-source strategy allows us to avoid the cost of iterative compilation and thus attain performance portability at low cost. Furthermore, we also propose a run-time system that simplifies code generation and realizes a novel programming model that handles execution parameters and local memory usage dynamically at run-time – something that has not been attempted before by any work that we know of.

2.5 Thesis Contributions

This thesis makes the following specific contributions with respect to source-to-source compilation for manycore processors :

1. The design and implementation of an object-oriented polyhedral compiler framework is presented, called *RosePolly*. Unlike similar work, RosePolly is designed as an API and therefore clearly distinguishes a set of extensible/customizable object-oriented building blocks for automatic parallelization and locality optimization of SCoPs.

2. A simple improvement to a well-known automatic parallelisation algorithm namely Pluto [BBK+08b] is proposed. This improvement enables us to distinguish between ambiguous dependence constraints that could lead to wavefront or fully parallel degrees of parallelism depending on their layout in the global constraint matrix.

3. A new source-to-source compilation path is proposed that produces parametrically tiled GPU code for static control programs (SCoPs). Such parameterized programs can be used as platform-independent templates amenable to run-time performance tuning. In the context of parametric GPU code generation this thesis makes the following individual contributions:

   - Develops a platform-independent run-time environment for effective mapping of wavefront parallelism to GPUs.
   - Proposes a simple loop vectorization algorithm that eliminates unecessary wavefront parallelism resulted from inter-statement dependences.
   - Introduces a dynamic local memory management scheme that enables run-time exploration of local memory usage through dynamic allocation of buffers.

4. Presents a preliminary investigation pertaining the feasibility of the parametric tiling run-time system as a general purpose programming model for GPUs. We call this the *Avelas* run-time system.

2.5.1 Publications

The research and technical contributions presented in this thesis resulted in the following publications :

1. *RosePolly: User's manual* – Technical documentation of the rosePolly framework publicly available as part of the ROSE compiler infrastructure[20] (`/rose/projects/RosePolly/doc`). (Chapter 5)

2. *More Definite Results from the Pluto Scheduling Algorithm* – presented at the first international workshop on polyhedral compilation techniques (IMPACT 2011) in conjunction with CGO 2011 (Chamonix France). (Chapter 4)

3. *Parametric GPU Code Generation for Static Control Programs* – International workshop on Languages and Compilers for Parallel Computing (LCPC 2013). (Chapter 6)

[20] `www.rosecompiler.org`

Part I

Analysis and Optimization

Chapter 3

The Polyhedral Model

This chapter presents an overview of the *polyhedral model*, an analysis and optimization framework for machine-independent automatic parallelization and locality optimization. After an introduction and overview of the framework the main components of polyhedral compilation are presented. These components are: (i) the domain (Section 3.3), (ii) the schedule or affine transformation (Section 3.4), (iii) the memory access functions (Section 3.5) and (iv) the polyhedral dependences (Section 3.6). Finally, Section 3.7 shows how these components can synthesize a practical polyhedral compiler while Section 3.8 provides a table (Table 3.1) with the most commonly used polyhedral libraries.

3.1 Introduction

The polyhedral model or affine transform theory [ALSU07] is an alternative to abstract syntax trees (AST) that enables compilers to analyse and transform programs by utilizing robust mathematical abstractions and libraries like systems of affine inequalities (i.e. Z-polyhedra), integer linear programming, Fourier-Motzkin elimination etc. The syntax trees that can be captured by the polyhedral model are called *Static Control Programs* or *SCoPs* and they are essentially loop-based computation kernels with statically predictable control flow. In practice, if we consider a high-level syntactic intermediate representation (IR) like the one used in ROSE

(see Section 2.2) then SCoPs are restricted to arbitrary-nested for-loops that comply to the SCoP restrictions of Section 2.3. However, Grosser et al [GGL12] showed that an SSA-based compiler like LLVM (see Section 2.2) can lift these syntactic restrictions by operating on a low-level representation where loop-based computations have a more generic definition.

One of the primary motivations for using the polyhedral model is to avoid a classic restriction of ASTs with respect to program transformations namely the phase ordering restriction [GVB+06]. In particular, multiple AST transformations are typically performed as ordered sequences of individual transformation steps each one relying on syntactic pattern-matching analysis. The ordering of these steps becomes an important concern if we consider their impact on the size and complexity of the AST. However, if we use a mathematical representation like the polyhedral model, this concern goes away since we can represent/compose arbitrary sequences of transformations that can be applied in a single step. The example of Figure 3.1 shows an ADI kernel (Alternating Direction Implicit) that has been analyzed and transformed using the polyhedral model. Notice that a combination of loop fusion and loop skewing has been applied to the original kernel simply through the affine functions of Figure 3.1b. These functions have been derived from a sophisticated polyhedral scheduling algorithm called Pluto [BR07, BBK+08b] that maximizes parallelism and locality through integer linear programming formulations.

3.2 Overview

The polyhedral representation of a *SCoP* (Static Control Program) consists of a list of computation statements $S_i : i = 0, \ldots, N$ each one carrying three main pieces of information namely the domain D_{S_i}, the schedule F_{S_i} and the access functions Π_{ij} for each memory reference $j = 0, \ldots, M_i$ of S_i. The domain D_{S_i} of each statement S_i denotes an execution domain through a finite set of affine inequalities i.e. a polyhedron. The schedule F_{S_i} denotes an execution order through a multi-dimensional affine transformation from the original domain D_{S_i} to a new transformed one[1] with a new lexicographic ordering. Evidently, finding schedules that lead to better software performance is the main objective of a polyhedral compiler. These schedules

[1] Polyhedral transformations are not restricted to unimodular ones [Ram95, Bas04]

```
for ( t = 0 ; t < TT ; t++ ) {
  for ( i1 = 0 ; i1 < NN ; i1++ ) {
    for ( i2 = 1 ; i2 < NN ; i2++ ) {
      S0(t,i1,i2) : X[i1][i2] = X[i1][i2] - X[i1][i2-1]*A[i1][i2]/B[i1][i2-1];
      S1(t,i1,i2) : B[i1][i2] = B[i1][i2] - A[i1][i2]*A[i1][i2]/B[i1][i2-1];
    }
  }
  for ( i1 = 1 ; i1 < NN ; i1++ ) {
    for ( i2 = 0 ; i2 < NN ; i2++ ) {
      S2(t,i1,i2) : X[i1][i2] = X[i1][i2] - X[i1-1][i2]*A[i1][i2]/B[i1-1][i2];
      S3(t,i1,i2) : B[i1][i2] = B[i1][i2] - A[i1][i2]*A[i1][i2]/B[i1-1][i2];
    }
  }
}
```

(a) Original program

$$F_{S_0} = (t + i1 + i2, t + i1, t + i2, 0)$$

$$F_{S_1} = (t + i1 + i2, t + i1, t + i2, 0)$$

$$F_{S_2} = (t + i1 + i2, t + i1, t + i2 + 1, 1)$$

$$F_{S_3} = (t + i1 + i2, t + i1, t + i2 + 1, 1)$$

(b) Affine functions

```
for ( c1=1 ; c1<=2*NN+TT-3 ; c1++ ) {
  if ( c1 <= NN-1 ) {
    S0(0,0,c1);
    S1(0,0,c1);
  }
  par for ( c2=max(1,c1-NN+1) ; c2<=min(c1-1,NN+TT-2) ; c2++ ) {
    if ( c2 <= NN-2 ) {
      S0(0,c2,c1-c2);
      S1(0,c2,c1-c2);
    }
    if ( c2 >= NN-1 ) {
      S0(c2-NN+1,NN-1,c1-c2);
      S1(c2-NN+1,NN-1,c1-c2);
    }
    par for ( c3=max(c1-NN+2,c1-c2+1) ; c3<=min(c1,c1-c2+TT-1) ; c3++ ) {
      S0(-c1+c2+c3,c1-c3,c1-c2);
      S1(-c1+c2+c3,c1-c3,c1-c2);
      S2(-c1+c2+c3-1,c1-c3+1,c1-c2);
      S3(-c1+c2+c3-1,c1-c3+1,c1-c2);
    }
    if ( c2 >= TT ) {
      S2(TT-1,c2-TT+1,c1-c2);
      S3(TT-1,c2-TT+1,c1-c2);
    }
  }
  if ( c1 <= NN+TT-2 ) {
    par for ( c3=max(1,c1-NN+2) ; c3<=min(TT,c1) ; c3++ ) {
      S2(c3-1,c1-c3+1,0);
      S3(c3-1,c1-c3+1,0);
    }
  }
}
```

(c) Transformed program

Figure 3.1: This example demonstrates the power of polyhedral compilation by showing how a composition of complex transformations like loop fusion and loop skewing is captured by a small set of affine functions.

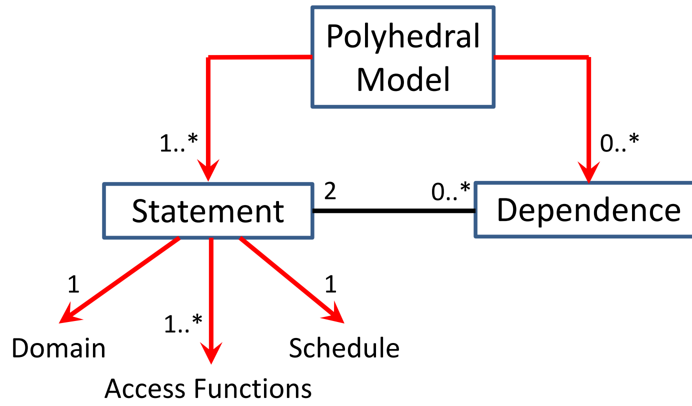


Figure 3.2: UML structure of the polyhedral model.

must respect a set of dependencies between memory accesses in order to ensure correctness of the transformed SCoP.

The next four sections will discuss these concepts in more detail with an emphasis on the data-structures used. Figure 3.2 depicts a UML view of the polyhedral model.

3.3 The Domain

Formally, if a statement S is surrounded by m loops then its execution domain is the convex set of all vectors $\vec{x}_S \in \mathbb{Z}_m$ that satisfy (3.1) where $\vec{x}_S$ is the iteration vector of S, $\vec{n}$ a vector of symbolic parameters and D_S an integer coefficient matrix where each row of D_S represents an affine loop-bound expression. Clearly, since iteration vectors can only have integral values, inequality (3.1) defines an integer polyhedron or in other words a Z-Polyhedron. The example of Figure 3.3 shows a simple loop nest along with a set of 6 affine inequalities corresponding to the loop bounds and conditionals surrounding S_0.

$$D_S \cdot \begin{pmatrix} \vec{x}_S \\ \vec{n} \\ 1 \end{pmatrix} \geq 0 \tag{3.1}$$

This set of affine inequalities can be used to construct the execution domain D_{S_0} of S_0 shown in Figure 3.3 (c). Note that the disjunctive conditional surrounding S_0 results in a concave

```
for ( i = 0 ; i < N ; i++ )
  for ( j = 5 ; j < N-5 ; j++ )
    if ( i >= 5 || j <= 10 )
      S0(i,j)
```

(a)

$$\begin{aligned} i &\geq 0 \\ N-1-i &\geq 0 \\ j-5 &\geq 0 \\ N-1-j &\geq 0 \\ i-5 &\geq 0 \\ 10-j &\geq 0 \end{aligned}$$

(b)

$$D_{S_0} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 \\ 0 & 1 & 0 & -5 \\ 0 & -1 & 1 & -1 \\ 1 & 0 & 0 & -5 \end{bmatrix} \xrightarrow{\text{OR}} \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 \\ 0 & 1 & 0 & -5 \\ 0 & -1 & 1 & -1 \\ 0 & -1 & 0 & 10 \end{bmatrix}$$

(c)

Figure 3.3: This example shows a syntactic form of a loop-nest (a), a finite set of affine inequalities corresponding to that loop-nest (b), and the actual polyhedral representation (c).

Z-Polyhedron which can be implemented as a list of integer matrices with each matrix representing a convex Z-Polyhedron according to (3.1) and the entire domain being a disjunctive list of such polyhedra.

3.4 The Schedule

The implicit execution order of all the points inside an execution domain is the lexicographic one. For example iteration (0, 4) is executed before (1, 0) according to the original execution domain of Figure 3.4. If we wish to change that order we would need to define an affine transformation that would map each point to a new transformed domain effectively changing the lexicographic order and as a result the execution order. Figure 3.4 shows an example of a loop skewing transformation implemented as an affine transformation while Figure 3.5 shows a loop fusion example.

Finding the right polyhedral transformations or schedules has been an active area of research since the early 90s. Early approaches relied on a space-time view of a schedule where two different algorithms were used for calculating a space mapping (machine independent parallelization or time minimization) and time mapping (machine dependent placement for communication minimization) respectively [Fea92b, Fea94, DV94, DR96]. Griebl [Gri04] later refined those

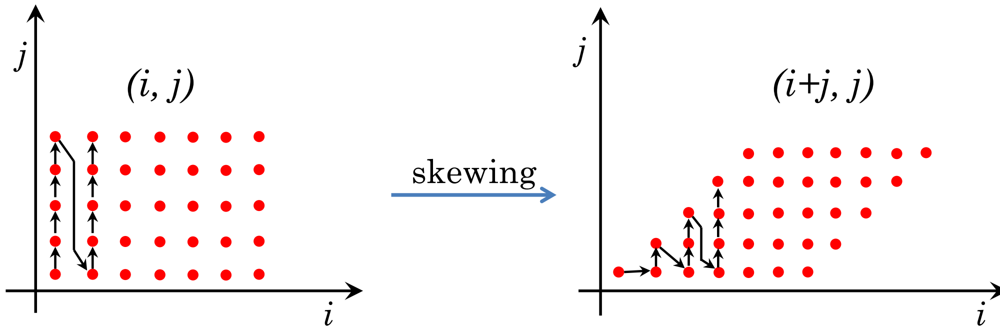


Figure 3.4: Loop skewing in the polyhedral model.

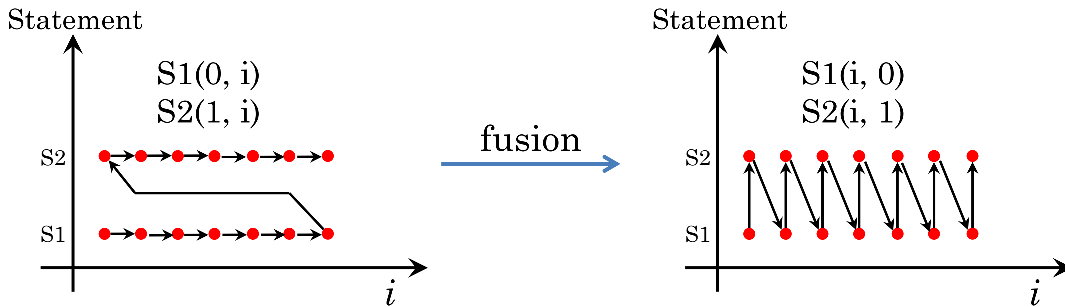


Figure 3.5: Loop fusion in the polyhedral model.

methods by introducing an index-set splitting technique – splitting a domain into multiple sections and deriving a schedule for each one separately – and a placement algorithm for distributed memory machines.

Lim et al [LCL99, LL98, ALSU07] introduced a holistic view of the problem with a single machine-independent scheduling algorithm that could find linearly independent partition mappings (i.e. affine transformations) that maximize the degree of parallelism and minimize communication at the same time. This idea was extended by Bondhugula et al [BBK+08b, BR07] with the introduction of the *Pluto* scheduling algorithm. A comprehensive and formal discussion of scheduling algorithms in the polyhedral model was recently presented by Darte et al [DRVV00]. It is worth noting that machine-dependent scheduling algorithms targeting GPUs have also been proposed by Baskaran et al [BBK+08a] and Cong et al [CZZ12]. The main motivation for these algorithms is the huge performance benefit we can get if we derive an execution schedule that yields coalesced global memory accesses. Later, Ueng et al [ULBH08] showed that such approach can be avoided if we use coalesced global memory accesses to promote uncoalesced data to local memory.

```
for ( i = 0 ; i < N ; i++ ) {
  for ( j = 0 ; j < N ; j++ ) {
    S0 : ... = A[i][j];
    S1 : A[i+1][j+1] = ...;
  }
}
```

(a) SCoP source

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} \vec{x}_{S_0} \\ N \\ 1 \end{bmatrix}$$

(b) Read access

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \vec{x}_{S_1} \\ N \\ 1 \end{bmatrix}$$

(c) Write access

Figure 3.6: Example showing how array access functions are captured by the polyhedral model

An orthogonal approach to scheduling was pursued by Girbal et al [GVB+06], Pouchet et al [PBB+10, PBB+11] and Vasilache et al [VBCG06] where schedules are derived through iterative compilation combined with empirical or model-driven search.

This thesis focuses on the Pluto scheduling algorithm [BBK+08b, BR07] or simply Pluto which is considered one of the most robust algorithms to date. A more detailed discussion will be presented in Chapter 4.

3.5 Memory Access Functions

Memory access functions are affine transformations that map each point of an execution domain to a point in the data space of the corresponding variable. Let Π_{ij} be an affine transform representing the memory access function of an array reference $j = 0, \ldots, N_i$ of a statement $S_i : i = 0, \ldots, M$. Figure 3.6 shows an example of memory access functions interpreted as affine transformation matrices. Note that scalar variables can follow this definition if we think of them as arrays with zero dimensionality.

3.6 Polyhedral Dependencies

A vital part of the polyhedral representation of a SCoP is a set E of polyhedral dependence edges $e \in E$. Polyhedral dependencies are vital because they comprise the main set of constraints that restrict the space of valid scheduling functions. Their main difference from conventional dependences [KA01, Wol90] is their ability to capture dependence edges between run-time

instances of statements thus giving us a fine-grained characterization.

A polyhedral dependence $e \in E$ is defined as a vector of affine functions $\vec{h}_e(\vec{x}_{sink_e})$ mapping a *sink* execution space characterized by $\vec{x}_{sink_e} \in D_{S_{sink_e}}$ to a *source* execution space characterized by $\vec{x}_{src_e} \in D_{S_{src_e}}$:

$$\vec{x}_{src_e} = (h_{e_0}(\vec{x}_{sink_e}), \ldots, h_{e_d}(\vec{x}_{sink_e}))$$

where d is the dimensionality of the *source* execution space. By putting those functions together with $D_{S_{sink_e}}$ and $D_{S_{src_e}}$ into a single integer matrix, we effectively form what is commonly known as the dependence polyhedron P_e of dependence $e \in E$. Figure 3.7 shows an example of a dependence polyhedron representing the true dependence between $S0$ and $S1$ of Figure 3.6. In particular, the upper left section captures $D_{S_{sink_e}}$, the section right below the upper empty section captures $D_{S_{src_e}}$ while the bottom section stores the mapping functions.

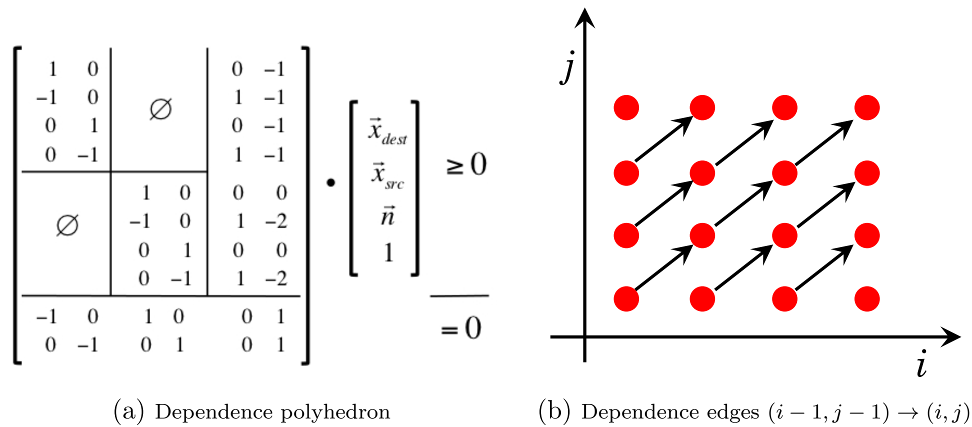
The dependence vector $e_v(e)$ associated with a dependence edge $e \in E$ can be defined by the following piecewise function:

$$e_v(e) = (\delta_{e_0}, \ldots, \delta_{e_d}),$$

$$\delta_{e_i} = \begin{cases} 1 & \text{if } \vec{x}_{sink_e} - h_{e_i}(\vec{x}_{sink_e}) > 0, \\ 0 & \text{if } \vec{x}_{sink_e} - h_{e_i}(\vec{x}_{sink_e}) = 0, \\ -1 & \text{if } \vec{x}_{sink_e} - h_{e_i}(\vec{x}_{sink_e}) < 0. \end{cases}$$

$$\text{for } i \in [0..d]$$

A dependence vector $e_v(e)$ can be implemented simply by appending each branch of δ_{e_i} to P_e and subsequently testing P_e for emptiness using a polyhedral library (see Section 3.8). Differences in the dimensionalities of $h_{e_i}(\vec{x}_{sink_e})$ and $\vec{x}_{sink_e}$ can be eliminated by adding semantics preserving one-time loops (i.e. dimensions) if necessary. The concept of dependence vectors will be useful



(a) Dependence polyhedron (b) Dependence edges $(i-1, j-1) \to (i, j)$

Figure 3.7: This example shows how the true dependence between Π_{00} and Π_{10} of Figure 3.6 is represented in the polyhedral model

in Chapter 6 as it provides a convenient abstraction of inter-tile dependences.

Polyhedral dependencies are derived from a dependence analysis algorithm which according to Pugh and Wonnacott [PW95] can be either *memory-based* or *value-based*. The former kind constructs a dependence edge for each pair of accesses that refer to the same memory location and at least one of them modifies that location (i.e. performs a write operation). Evidently, this approach is conservative as each access depends on the last write operation as opposed to all preceding write operations. Therefore, this may lead to redundant dependences[2] that can degrade the performance of a scheduling algorithm (as more dependences correspond to more optimization constraints for the scheduling algorithm to satisfy), but without affecting the derived schedules. Nevertheless, memory-based dependence analysis is easier to implement and faster to execute thus being a sensible approach in practice.

The latter kind (i.e. value-based dependence analysis) eliminates the redundant memory-based dependences and keeps only those that don't have intermediate writes. In other words it offers precise dependence analysis also known as *array data-flow analysis*. The seminal work on value-based dependence analysis was done by Pugh and Wonnacott [PW94] and Feautrier [Fea91], while Griebl [Gri04] presented a pragmatic summarization and discussion along with a practical dependence analysis algorithm.

[2]These dependences are also called *transitive dependences*

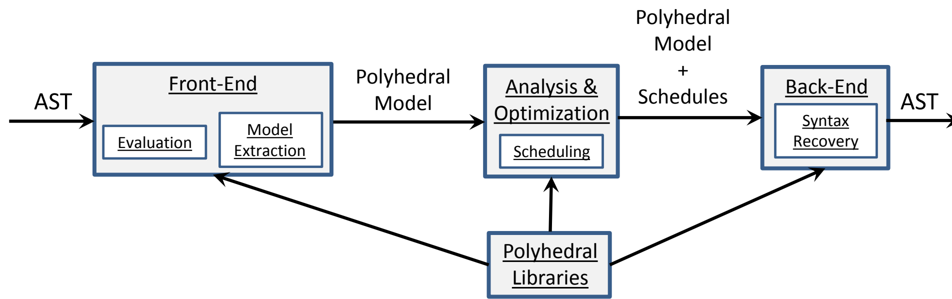


Figure 3.8: Basic polyhedral compilation flow

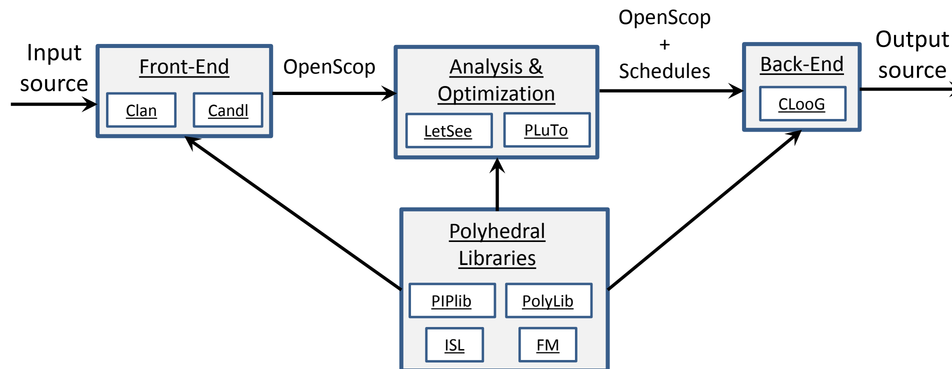


Figure 3.9: Compilation flow and structure of the Polyhedral Compiler Collection or PoCC

3.7 Compilation Flow

A typical polyhedral compilation flow consists of three main stages. The first stage (front-end) is responsible for extracting the polyhedral model (including dependence polyhedra) from a syntactic representation of the program. The second stage optimizes the program and derives a new execution order through affine transformations (i.e. scheduling) as described in Paragraph 3.4. Finally, the last stage (back-end) converts the polyhedral model back to a syntactic tree or directly into the target source code. Figure 3.8 depicts this structure, while Figure 3.9 shows how this generic flow is reflected on a real polyhedral compiler called PoCC[3].

3.8 Polyhedral Libraries

What drives a polyhedral compilation engine is a set of polyhedral library functions providing fundamental abstractions and operations on systems of affine constraints like intersection,

[3]http://pocc.sourceforge.net

Library	Description
PolyLib	Collection of binary and unary operations on vectors, matrices, lattices, polyhedra, Z-Polyhedra and unions of polyhedra. Does not include ILP solvers (http://icps.u-strasbg.fr/polylib/).
PIPlib	PIP stands for Parametric Integer Programming and is a library used for solving parametric integer linear programs. It is used in order to find the lexicographic minimum (or maximum) in a set of integral points belonging to a convex polyhedron (http://www.piplib.org/).
PPL	The Parma Polyhedral Library provides abstractions for handling polyhedra along with a rich set of operations. It also provides a parametric integer programming solver based on an exact-arithmetic version of the simplex algorithm (http://bugseng.com/products/ppl/).
Omega	Similar to PPL and Polylib (http://www.cs.umd.edu/projects/omega/).
ISL	Inspired by the Omega system, the Integer Set Library provides a complete set of abstractions and operations on systems of affine constraints. It also includes ILP solvers with min/max operations on polyhedra and an intuitive front-end interface/parser system that enables users to utilize the library through a high-level language (https://www.ohloh.net/p/isl).
CLooG	It provides an extended qulere et al. [QRW00] algorithm for efficiently scanning polyhedra with for-loops. In other words it provides the necessary functionality for converting the polyhedral model back to a syntactic form (http://www.cloog.org/).

Table 3.1: List of polyhedral libraries accompanied with a short description.

union, projection along with integer linear programming solvers (ILP), Fourier-Motzkin elimination etc. Table 3.1 presents a set of the most commonly used libraries with a short description.

Chapter 4

Automatic Parallelization: The Pluto Scheduling Algorithm

The *Pluto* scheduling algorithm [BBK+08b, BR07] is a well-known algorithm for automatic parallelization and locality optimization in the polyhedral model and is considered the default machine-independent loop optimizer for the remainder of this thesis. It seeks linearly independent affine transformations – or schedules – for each statement of a SCoP, such that total communication is minimized[1]. This is achieved by constructing and solving systems of affine constraints on schedule coefficients by means of integer linear programming (ILP) solvers. In this chapter we are going to outline the basic concepts and architecture of the Pluto algorithm and also show that in some cases Pluto can be sensitive to the layout of the scheduling constraints. More specifically, the sensitivity lies in the ordering of the unknown coefficients in the constraint matrices or in other words in the ordering of their columns.

[1] Informally, in the context of Pluto scheduling, communication is the number of distinct loop iterations that depend on a given loop iteration

4.1 Pluto Scheduling

A 1-D affine transform of a statement S denoted by $\Phi(\vec{x}_S) = h \cdot \vec{x}_S$ – where $\vec{x}_S$ is the iteration vector of S and h a row vector – maps each runtime instance of S to a new hyperplane instance on the transformed iteration space. Two run-time instances $\vec{x}_S^1$ and $\vec{x}_S^2$ of S where $\Phi(\vec{x}_S^1) = \Phi(\vec{x}_S^2)$ belong to the same hyperplane instance or in other words to the same loop iteration of the transformed iteration space. Therefore, $\Phi(\vec{x}_S)$ effectively represents a new loop in the transformed space.

Obviously, in order to obtain such transforms we need to make sure that they do not violate any of the dependencies E of the original program. In other words we need to make sure that for each dependence $e \in E$ the sink run-time instance $\vec{x}_{sink} \in P_e$ is mapped to the same or subsequent hyperplane instance than the source $\vec{x}_{src} \in P_e$. In the Pluto context these are called *permutability constraints* or *legality-of-tiling constraints* and are formulated as follows:

$$\Phi(\vec{x}_{sink}) - \Phi(\vec{x}_{src}) \geq 0, \qquad \forall \vec{x}_{sink}, \vec{x}_{src} \in P_e, \quad \forall e \in E \tag{4.1}$$

An additional set of constraints comes from the need to minimize the distance between the source and the sink of each dependence or in other words minimize communication. This is done by introducing a cost function $\delta_e(\vec{n})$ to (4.1) which is an unknown affine expression on the symbolic parameters $\vec{n}$ of our SCoP. As a result we get the so called *communication bounding constraints* that are formulated as follows:

$$\delta_e(\vec{n}) \geq \Phi(\vec{x}_{sink}) - \Phi(\vec{x}_{src}), \qquad \forall \vec{x}_{sink}, \vec{x}_{src} \in P_e, \quad \forall e \in E \tag{4.2}$$

A crucial observation is that both (4.1) and (4.2) are not affine constraints simply because the coefficients of both $\Phi(\vec{x})$ and $\delta_e(\vec{n})$ are unknown. In fact these are the coefficients we are looking for thus we need to eliminate the iteration vectors $\vec{x}_{sink}$ and $\vec{x}_{src}$ and the vector of symbolic parameters $\vec{n}$ from (4.1) and (4.2). In order to do that, Pluto utilizes the affine form of a well known result from linear programming namely Farkas' lemma [Sch98]. According to

Feautrier [Fea92a] Farkas' lemma states that an affine expression $\psi(\vec{x})$ is positive everywhere in a domain D_ψ if and only if it is the sum of the faces of D_ψ (i.e. individual inequalities or rows) each one multiplied by a non-negative multiplier. More specifically we have:

$$iff \quad \psi(\vec{x}) \geq 0 \quad \text{everywhere in} \quad D_\psi \quad \text{then} \quad \psi(\vec{x}) \equiv \lambda_0 + \sum_k \lambda_k \cdot D_\psi, \quad \text{for} \quad \lambda_0, \lambda_k \geq 0 \tag{4.3}$$

We can now use Farkas lemma on (4.1) and (4.2) and get the following identity relations:

$$\Phi(\vec{x}_{sink}) - \Phi(\vec{x}_{src}) \equiv \lambda_{e0} + \sum_k \lambda_{ek} \cdot P_e, \quad \text{for} \quad \lambda_0, \lambda_{ek} \geq 0 \tag{4.4}$$

$$\delta_e(\vec{n}) + \Phi(\vec{x}_{src}) - \Phi(\vec{x}_{sink}) \equiv \lambda_{e0} + \sum_k \lambda_{ek} \cdot P_e, \quad \text{for} \quad \lambda_0, \lambda_{ek} \geq 0 \tag{4.5}$$

Through simple algebraic manipulation – i.e. identification – we can now eliminate $\vec{x}_{src}$, $\vec{x}_{sink}$ and $\vec{n}$ from (4.4) and (4.5). Because each equality can be written as a pair of inequalities, after the identification step and after eliminating all the farkas multipliers from (4.4) and (4.5) using Fourier-Motzkin elimination, we end up with two sets of constraints which are the following:

$$C_e^p \cdot \vec{y} \geq 0, \quad \text{where} \quad \vec{y}^\top = [\overbrace{a_1 \quad \cdots \quad a_n \quad c}^{\text{Schedule Coefficients}}] \tag{4.6}$$

$$C_e^c \cdot \vec{y} \geq 0, \quad \text{where} \quad \vec{y}^\top = [\overbrace{p_1 \quad \cdots \quad p_m \quad w}^{\text{Cost Coefficients}} \quad a_1 \quad \cdots \quad a_n \quad c] \tag{4.7}$$

where n is the number of dimensions for $\vec{x}_{sink}$ (i.e. surounding loops), m the number of symbolic parameters in our SCoP and w and c coefficients of constant terms. These constraints can be easily combined into a global constraint matrix C_{global} that would also accumulate the constraints from all dependence edges $e \in E$:

$$C_{global} \cdot \vec{y} \geq 0, \quad \text{where} \quad \vec{y}^{\top} = [\overbrace{p_1 \quad \cdots \quad p_m \quad w}^{\delta_e(\vec{n})} \quad \overbrace{a_{11} \quad \cdots \quad a_{1n_1} \quad c_1}^{\Phi_{S_1}(\vec{x}_{S_1})} \quad \overbrace{\cdots}^{\cdots} \quad \overbrace{a_{N1} \quad \cdots \quad a_{Nn_N} \quad c_N}^{\Phi_{S_N}(\vec{x}_{S_N})}] \tag{4.8}$$

where N is the total number of statements in our SCoP and n_i the dimensionality of statement $i \in [1..N]$.

An integer linear programming solver (e.g. PIP [Fea88]) can now be used to acquire the lexicographic minimum for $\vec{y}$ in (4.8). Since the coefficients of $\delta_e(\vec{n})$ are in the leading minimization positions the derived schedule coefficients are the ones that yield minimum communication i.e. minimum coefficients for $\delta_e(\vec{n})$. The Pluto algorithm though does not stop here because we actually seek multi-dimensional schedules for each statement of our SCoP (i.e. at least as many solutions for each statement as its dimensionality). Therefore, after getting the first solution, Pluto appends the so called *orthogonality constraints* to the global constraint matrix C_{global} to make sure that all subsequent solutions are linearly independent.

In order to construct the orthogonality constraints Pluto first derives the orthogonal sub-space $H_S^{\perp}$ of the solutions found so far (matrix H_S) for a statement S which is defined as follows [LP93, Pen55]:

$$H_S^{\perp} = I - H_S^{\top}(H_S \cdot H_S^{\top})^{-1} \cdot H_S \tag{4.9}$$

After obtaining the orthogonal sub-space $H_S^{\perp}$ we need to make sure that the new solution has a non-negative component in $H_S^{\perp}$. Pluto does that by appending the following constraints to C_{global}:

$$\forall i, H_S^{i\perp} \cdot \vec{h}_S^{new} \geq 0 \qquad \wedge \qquad \sum_i H_S^{i\perp} \cdot \vec{h}_S^{new} \geq 1 \tag{4.10}$$

where i denotes individual rows of $H_S^{\perp}$ and $\vec{h}_S^{new}$ the coefficients of the new schedule dimension for S. These constraints essentially state that there must be at least one row (i.e. basis vector)

of the orthogonal sub-space $H_S^{\perp}$ for which the dot product with the new solution is non-zero and positive. Restricting the dot product to be positive is done for efficiency because taking negative solutions into account would yield a combinatorial explosion that has no practical benefit. Evidently, these constraints ensure linear independence between the new solution and the already found ones.

Pluto also adds an extra set of constraints to avoid the trivial solution of all schedule coefficients being zero. These are called *non-trivial solution constraints* and simply enforce $\Phi(\vec{x_S}) \geq 1$ for each statement S.

Algorithm 1 presents the core structure of the Pluto scheduling algorithm – the result of putting together all the components we have described so far. As we see the Pluto algorithm finds linearly independent affine transforms (i.e. schedules) and stops when $max(n_i)$ – where n_i is the dimensionality of statement S_i – solutions have been found and all dependencies are killed. A dependence is killed by a 1-D affine transform $\Phi(\vec{x})$ if the following condition holds:

$$\Phi(\vec{x_{dest}}) - \Phi(\vec{x_{src}}) > 0 \tag{4.11}$$

If the condition in line 15 fails then the algorithm removes all satisfied dependencies so far and tries again. If no solution was found in line 15 (i.e. $bandSols = 0$) then the algorithm attempts to cut the dependence graph of the program into strongly-connected components (i.e. *scc*) and add scalar dimensions to the schedules H before removing any killed dependencies. These scalar dimensions correspond to the topological sort of the scc components.

In the next section we will show that sometimes the results we get from the ILP solver are sensitive to the minimization order or in other words to the layout of $\vec{y}$ in (4.8).

Algorithm 1 The core structure of the Pluto scheduling algorithm. The input E is the set of polyhedral dependence edges of our SCoP.

```
1:  procedure PLUTO(E)
2:      totalSols ← 0                                   ▷ Total number of solutions is zero
3:      H ← 0                                           ▷ Solution matrix initialized
4:      ddg ← E                                         ▷ Construct data-dependence graph from E
5:      ndeps ← |E|                                     ▷ Number of unsatisfied dependences
6:      while (totalSols < max(n_i)) OR (ndeps ≠ 0) do
7:          C_global ← 0                                ▷ Initialize global constraint matrix
8:          C_global ← legality-of-tiling(E)            ▷ Append permutability constraints
9:          C_global ← Communication(E)                 ▷ Append communication bounding constraints
10:         C_global ← non-trivial                      ▷ Append non-trivial solution constraints
11:         bandSols ← 0                                ▷ Counts the number of solutions in a band
12:         if (H ≠ 0) then                             ▷ At least one solution has already been found
13:             C_global ← orthogonality(H)             ▷ Append orthogonality constraints
14:         end if
15:         while (lexMin(C_global)) do                 ▷ Invoke ILP solver
16:             bandSols ← bandSols + 1
17:             H ← solution                            ▷ Append solution to H
18:             C_global ← orthogonality(H)             ▷ Append orthogonality constraints
19:         end while
20:         totalSols ← totalSols + bandSols            ▷ Update global solutions counter
21:         if (bandSols = 0) then
22:             CutSCC(ddg,H)                           ▷ Cut into strongly-connected components and update H
23:         end if
24:         updateDDG(H,ddg,E,ndeps)                    ▷ Remove killed dependences from ddg and update ndeps and E
25:     end while
26: end procedure
```

4.2 Resolving Ambiguous Constraints

Motivating Example

The problem manifests itself when there is a situation in which we have the same communication cost $\delta_e(\vec{n})$ (see 4.2) for more than one solutions therefore the minimization algorithm will pick a solution according to the ordering of the schedule coefficients. The example of Figure 4.1 shows two schedules for statement S_0 that even though both have the same degree of parallelism the second one has a fully parallel loop as opposed to the wavefront-parallel[2] execution space of the first schedule.

First of all, by laying out the constraints from both dependencies we realize that at the beginning there is no possible solution that has zero communication i.e. there is no fully parallel loop.

[2] A wavefront-parallel execution space has no parallel loops per-se, but parallelism is possible through a loop-skewing transformation. Such transformation will reveal an inner parallel loop. The price we pay is the start-up and drain cost associated with the non-rectangular shape of the skewed iteration space.

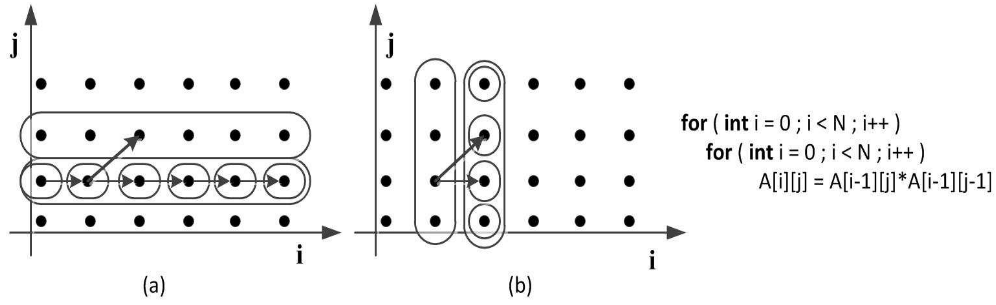


Figure 4.1: Schedule (a) is $\Phi(x_{S_0}) = (j, i)$ and results in wavefront parallelism while schedule (b) is $\Phi(x_{S_0}) = (i, j)$ and results in one fully parallel loop.

Therefore at the first iteration of the algorithm the minimum communication cost is 1 and can be obtained by two solutions: $\Phi(\vec{x_{S_0}}) = i$ and $\Phi(\vec{x_{S_0}}) = j$ i.e. minimum coefficients for i and j for communication cost 1. By putting the coefficient of i (a_i) before that of j (a_j) in the global constraint matrix the ILP solver will minimize a_i first, giving as the answer $a_i = 0, a_j = 1$ or $\Phi(\vec{x_{S_0}}) = j$ as the first solution. By adding the orthogonality constraints we then get the $\Phi(\vec{x_{S_0}}) = i$ as our second linearly independent solution. If we now reverse the order of a_i and a_j we will get $\Phi(\vec{x_{S_0}}) = i$ and $\Phi(\vec{x_{S_0}}) = j$. From Figure 4.1 we see that the order in which we get these two solutions matters since Figure 4.1-(b) presents a sequential loop followed by a parallel one while Figure 4.1-(a) presents a wavefront-parallel loop nest.

Of course, one cannot be certain about which one of the two possible solutions will turn out to be better in practice. It is very likely that fully parallel loops will perform better in most cases since wavefront parallelism comes with a start-up and drain cost. However, depending on the problem sizes, wavefront parallelism might end up being equally good or even better if it has better temporal or spatial locality along its wavefronts. In the next paragraph we show a simple method to get the right order for the constraint coefficients that takes wavefront cost and temporal locality into account.

Proposed Solution

The reason why the scheduling algorithm is unable to distinguish between these two solutions is because both dependencies in Figure 4.1 have the same communication cost along each dimension i and j. The difference between them lies on their direction i.e. one of the dependencies

extends into both i and j dimensions as opposed to the other one that extends along i only (carried by only one of the two loops). By extracting and using this information we could be able to determine the right order for the constraint coefficients and distinguish between pipeline and fully parallel degrees of parallelism.

Let S_{dest} be the destination statement of a dependence edge $e \in E$. We define a bit vector $\vec{V_e}$ with size $min(m_{dest}, m_{src})$ (dimensionalities of S_{dest} and S_{src}) that would store the direction information for e. We also store a boolean attribute H_e which is false if a dependence vector extends along more than one dimension. In particular :

$$\vec{V_e}[i] = \begin{cases} 1 & \text{if } e \text{ extends along } i, \\ 0 & \text{if } e \text{ doesn't extend along } i \end{cases}, \quad 0 \le i < min(m_{dest}, m_{src}) \tag{4.12}$$

$$H_e = \begin{cases} true & \text{if } e \text{ is horizontal}, \\ false & \text{if } e \text{ is diagonal} \end{cases} \tag{4.13}$$

Each dependence edge $e \in E$ is represented by a dependence polyhedron P_e defined as follows :

$$P_e = \begin{array}{c} L \\ m_{src} \end{array} \left[\begin{array}{c|c|c} D_{dest} & \varnothing & \\ \hdashline \varnothing & D_{src} & \\ \hline \multicolumn{3}{c}{h_transformation} \end{array} \right] \cdot \begin{bmatrix} \vec{x}_{S_{dest}} \\ \vec{x}_{S_{src}} \\ \vec{n} \\ 1 \end{bmatrix} \begin{array}{c} \ge 0 \\ \overline{} \\ = 0 \end{array} \tag{4.14}$$

(column widths of the matrix: m_{dest}, m_{src}, $(n+1)$)

By taking 4.14 into account we can use Algorithm 2 to populate the direction vectors for each $e \in E$.

Algorithm 2 Direction extraction

```
1: for each e ∈ E do
2:     V_e ← 0
3:     bool H_e ← true
4:     int count ← 0;
5:     for i = 0 to min(m_dest, m_src) do
6:         if P_e[L + i][m_dest + i] + P_e[L + i][i] = 0 then
7:             if ∃j ≠ i, (m_dest + i) s.t. P_e[L + i][j] ≠ 0 then
8:                 V_e[i] ← 1;
9:                 count++;
10:            end if
11:        else
12:            count++;
13:        end if
14:    end for
15:    if count > 1 then
16:        H_e ← false
17:    end if
18: end for
```

Upon construction of the global constraint matrix we can determine the order of the transform coefficients for each statement using Algorithm 3.

Algorithm 3 Coefficient ordering algorithm

```
1: Let N be the total number of statements in the source program
2: for  each Statement S_i, 0 ≤ i < N do
3:     Let V_{S_i} a bit vector with size m_{S_i} initialized to 0
4:     for  each e ∈ E s.t. S_dest = S_i do
5:         if H_e = true then
6:             V_{S_i} = V_{S_i} OR V_e
7:         end if
8:     end for
9:     for  each element j of V_{S_i} do
10:        if V_{S_i}[j] = 0 then
11:            Put coefficient a_{S_j} in leading minimization position
12:        end if
13:    end for
14: end for
```

In our example we have two dependence edges e_1 and e_2 where $V_{e_1} = [1, 1]$ and $V_{e_2} = [1, 0]$. Furthermore, the first edge e_1 is diagonal so $H_{e_1} = false$ and $H_{e_2} = true$. Therefore, Algorithm 3 will give us $V_{S_0} = [1, 0]$ and as a result we will put a_{S_j} in the leading minimization position.

By applying this technique we can choose fully parallel degrees of parallelism instead of pipeline ones. However, as we already mentioned this might not be the best strategy depending on problem sizes and locality along a wavefront. A wavefront for statement S on an m-dimensional loop nest can be represented by the following schedule:

$$\Phi_{wave_S}(\vec{x_S}) = \overbrace{\begin{bmatrix} 1 & 1 & \ldots & 1 \end{bmatrix}}^{m} \cdot \vec{x_S} \tag{4.15}$$

We can measure the volume of temporal locality within a wavefront by counting the Read-after-Read (input) dependencies that satisfy the following condition :

$$\Phi_{wave_{S_{dest}}}(\vec{x_{S_{dest}}}) = \Phi_{wave_{S_{src}}}(\vec{x_{S_{src}}}) \tag{4.16}$$

We can then define empirical thresholds for the structure parameters and the temporal reuse along a wavefront to decide whether pipeline parallelism would be better for a particular hardware architecture or not. Deriving these empirical thresholds for different architectures requires experimental investigation that could be subject for future research.

4.2.1 Conclusions

In this section it was shown that a widely-used polyhedral scheduling algorithm for automatic parallelization [BBK+08b, BR07] can sometimes be sensitive to the layout of the global constraint matrix that we use to obtain our solutions. To overcome this ambiguity we propose an empirical methodology based on the direction of each dependence vector that tries to find the right order for the unknown constraint coefficients. The right order assumes that a fully parallel degree of parallelism is usually better than wavefront parallelism. However, we showed that the volume of temporal reuse along a wavefront can be calculated enabling us to derive empirical machine-dependent thresholds to make a more precise decision.

Chapter 5

RosePolly:

Design and Implementation of an Object-Oriented Polyhedral Framework

This chapter introduces *RosePolly*, a novel polyhedral compilation framework based on the *ROSE*[1] compiler infrastructure. Unlike existing frameworks, RosePolly is designed as an object-oriented API as opposed to a monolithic executable. This API is organized into three layers of abstraction each one corresponding to a separate conceptual layer of the polyhedral model. These layers are: (i) the compilation layer (Section 5.3), (ii) the polyhedral model layer (Section 5.5) and (iii) the math layer (Section 5.6). In this chapter we are going to look into each one of these layers separately and how all of them fit together into a uniform design that encourages modular and customized use of polyhedral compilation technologies.

[1] www.rosecompiler.org

5.1 Polyhedral Compilation in Practice: Related Work and Motivation

Loopo[2] and *SUIF*[3] were among the first practical polyhedral compilers that appeared in the mid 90s. Both of them offered automatic parallelization for affine loop nests, i.e. SCoPs, through their automatic scheduling algorithms [Len93, LCL99]. However, code generation in the polyhedral model (i.e. converting scheduled polyhedra back into loops) was one of the major combinatorial bottlenecks at the time that impeded these frameworks (and polyhedral compilation in general) from being more widely used and researched.

It was not until the emergence of *CLooG*[4] in the mid 2000s that loop optimization in the polyhedral model started to become more practical and attractive. CLooG was the first robust and freely available tool for code generation in the polyhedral model that later became the key component in most polyhedral compilers. It is based on the Quillere et al. [QRW00] algorithm for generating efficient loop nests from polyhedra but improves it by applying techniques for avoiding code explosion and complexity issues.

One of the first attempts to utilize the CLooG code generation technology was the *Pluto* compiler [BR07]. Pluto is a robust and practical framework that implements the Pluto scheduling algorithm (see Chapter 4) for automatic parallelization and locality optimization of SCoPs. Pluto was shown to generate effective parallel and tiled code for SMP systems through OpenMP and was later adopted by a well-known industrial compiler namely IBM XL compiler. However, Pluto is a stand-alone source-to-source compiler that was not designed to be extensible. This was the main design goal of *PoCC*[5].

The *PoCC* source-to-source compiler was designed to be a modular and extensible framework that could connect pluggable independent modules together through a common intermediate representation called *ScopLib* (recently replaced by an improved standard called *OpenScop*).

[2] www.infosun.fmi.uni-passau.de/cl/loopo
[3] suif.stanford.edu/
[4] www.cloog.org
[5] www.cs.ucla.edu/~pouchet/software/pocc

The default PoCC modules include CLooG and Pluto as well as *Clan* for model extraction, *Candl* for dependence analysis and *LetSee* [PBB+10] – an alternative polyhedral optimizer that seeks optimal schedules through searching structured optimization spaces – as shown in FIgure 3.9. PoCC was the first attempt of the polyhedral community to formulate a software hub that would bring together independent projects on polyhedral compilation.

Polly[6] was the first attempt to implement a polyhedral compiler as an LLVM optimization pass. As such, Polly was designed to operate on an SSA intermediate form (i.e. LLVM-IR) which eliminated important syntactic restrictions. However, we believe that since polyhedral optimizations are primarily syntactic in nature (e.g. loop transformations and loop parallelization), operating on an SSA form is a counter-intuitive strategy especially when it comes to code generation for heterogeneous architectures like GPUs. Targeting such architectures can be significantly easier and more portable if we operate on a higher syntactic level that leverages programming models like CUDA. It is worth noting that a very similar framework was implemented as a GCC optimization pass called *GRAPHITE* [TCE+10]. Just like Polly, GRAPHITE operates on an SSA form called *GIMPLE* and as a result suffers from the same code generation weakness in our opinion.

Perhaps the most robust and widely used polyhedral frameworks today is *ISL* [Ver10]. ISL started as a polyhedral library offering functions for manipulating integer polyhedra. However, ISL today has been extended to support compilation operations like dependence analysis, scheduling – providing Pluto and Feautrier [Fea92b] scheduling options – and code generation. In fact, ISL was used recently to develop a state-of-the-art source-to-source compiler for automatic C-to-CUDA code generation for SCoPs [VCJC+13]. Furthermore, unlike most frameworks, ISL was designed to be used as an API which facilitates easier and customized development of polyhedral compilers and tools.

Efforts to realize production-quality polyhedral compilers were also made by Amini et al. [ACE+12] and Leung et al. [LVM+10]. Both approaches support well-known automatic scheduling algorithms like Pluto and Feautrier and code generation for heterogeneous architectures like GPUs

[6] polly.llvm.org

based on the CLooG code generation technology.

In this chapter we introduce *RosePolly*, an object-oriented polyhedral framework that was designed to be a user-centric API that facilitates easy and customized compiler and tool development – something very useful for our main research objectives. To the best of our knowledge this is the first attempt to realize such an API for polyhedral compilation. By taking advantage of object-oriented design features like inheritance and polymorphism we believe that extending and/or customizing existing polyhedral compilation technologies – or implementing new ones – can become much easier. Such capability will allow users to reach their research objectives faster and also acquire a better understanding of polyhedral compilation and what each individual building block can do.

5.2 Overview: The 3-Layer Interface

RosePolly is an object-oriented polyhedral compilation API based on the *ROSE*[7] compiler infrastructure. As a result it operates on a high-level syntactic intermediate representation (IR) produced by the production-quality EDG[8] front-end of ROSE. Such syntax-based IR allows us to formulate source-to-source code generation methods that can leverage high-level languages like CUDA for efficient GPU code.

Perhaps one of the most important properties of RosePolly is its design. In particular, it is organized into three layers of abstraction as shown in the UML diagram of RosePolly (Figure 5.1) and in Figure 5.2 as a bottom-up stratification. With this design we can effectively separate the polyhedral model (the new IR of our program) from the high-level compilation primitives – like model construction, scheduling and code generation – and the low-level math primitives like integer linear programming, operations on integer polyhedra etc. We believe that this is a very good software engineering practice that allows us to maintain each layer separately without affecting the other ones. For example, notice from Figure 5.1 that adding a new compilation primitive (e.g. a new scheduling or code generation algorithm) is as simple

[7] www.rosecompiler.org
[8] www.edg.com

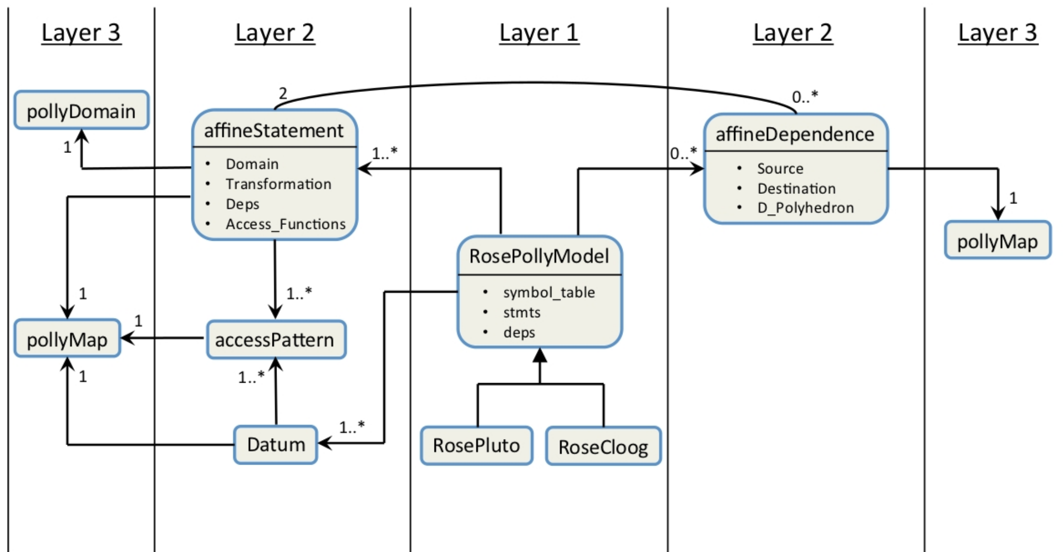


Figure 5.1: UML structure of RosePolly

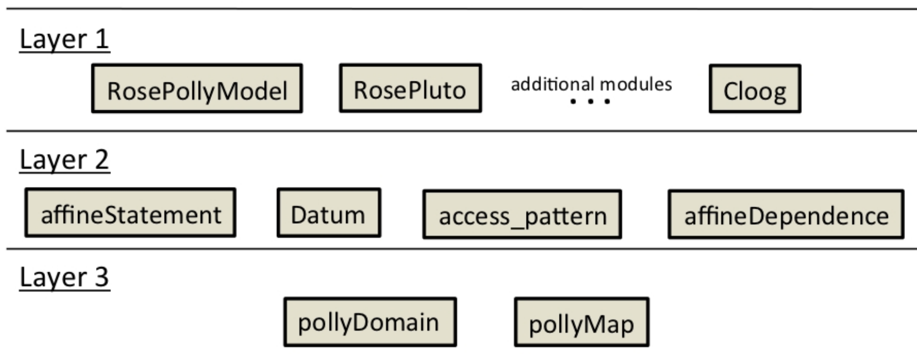


Figure 5.2: RosePolly structure as a bottom-up stratification of abstraction layers.

as implementing a new `RosePollyModel` sub-class that will utilize existing layer-1 and layer-2 functionality through well-defined interfaces. In the following sections we are going to look into each layer in more detail (Sections 5.3-5.6).

5.3 Layer-1 – The Compilation Interface

This layer encompasses the high-level analysis and optimization passes that can be used to construct a meaningful source-to-source compilation flow. Prerequisite to any of these passes though is a valid polyhedral model of the input program which is captured by the `RosePollyModel` class. Consequently, the first major component of layer-1 is the model extraction function or

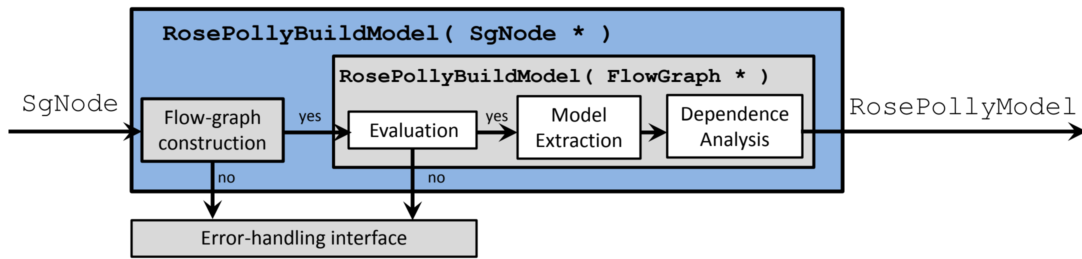


Figure 5.3: The structure of the RosePolly model-extraction function.

in other words the function responsible for constructing `RosePollyModel` objects from ROSE syntax trees. This function is depicted in Figure 5.3 and takes an `SgNode` pointer as input – i.e. a pointer to a ROSE syntax-tree – and returns a `RosePollyModel` object provided that a control-flow graph of the syntax tree can be constructed and the SCoP restrictions are not violated.

The control-flow graph or CFG of the input program is an essential data-structure in this process. In the context of RosePolly, the CFG is implemented by the `FlowGraph` class and consists of three main types of nodes: the `ForLoop` node, the `Conditional` node and the `Statement` node. Figure 5.4 shows an example of a SCoP with its corresponding `FlowGraph` structure. Notice that `ForLoop` and `Conditional` nodes in the `FlowGraph` can be of type *head* or *tail*. This feature can be very useful for `FlowGraph` traversals because it allows us to easily navigate through the divergent control paths from either direction, i.e. forwards or backwards.

After obtaining a `FlowGraph` the `RosePollyBuildModel` method (Figure 5.3) evaluates it against the SCoP restrictions of Section 2.3. This is done through a visitor-pattern traversal of the `FlowGraph`. This traversal visits each node and invokes the right evaluation procedure according to its type. The same mechanism is used for translating `FlowGraph` nodes into the polyhedral model once the evaluation process succeeds. The per-node-type procedures for evaluation and model extraction are implementations of the `RosePollyCustom` interface defined as follows:

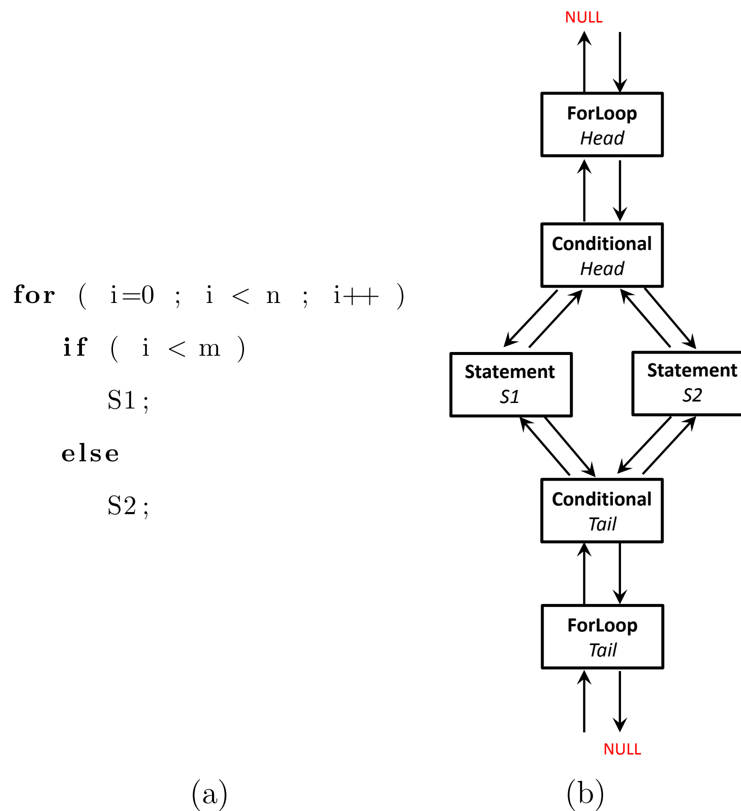


Figure 5.4: The `FlowGraph` (b) of a simple SCoP (a). Each `ForLoop` and `Conditional` node can be of type *Head* or *Tail*. This allows us to traverse the `FlowGraph` easily from any direction.

```
class RosePollyCustom {
    public:
    virtual bool evaluate_loop( ForLoop * loop ) =0;
    virtual pollyDomain * add_loop( pollyDomain * d, ForLoop * loop ) const=0;


    virtual bool evaluate_conditional( Conditional * cond ) const=0;
    virtual pollyDomain * add_conditional( pollyDomain * d, Conditional * cond ) const=0;


    virtual bool evaluate_access( AccessPattern * ap ) const=0;
    virtual pollyMap * add_pattern( pollyDomain * m, AccessPattern * ap ) const=0;


    virtual void add_params( vector<string> p ) =0;
}
```

The `AccessPattern` objects are attributes of the `Statement` node which is the base class of the `affineStatement` node shown in the UML diagram of Figure 5.1. Therefore, every time a

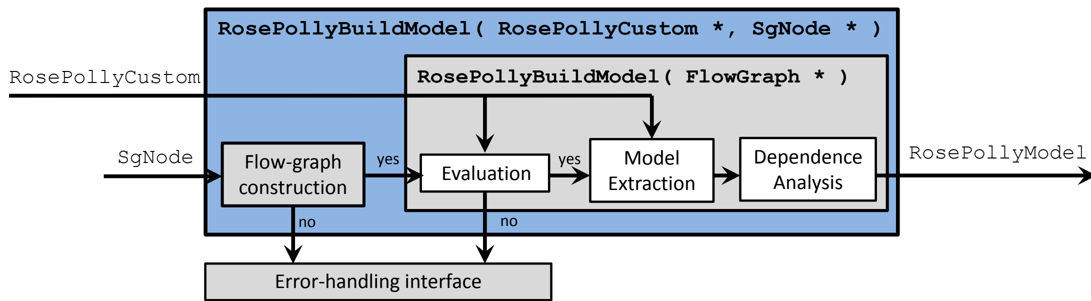


Figure 5.5: The overloaded version of the `RosePollyBuildModel` method that takes an additional `RosePollyCustom` argument that implements the evaluation and model extraction policies.

`Statement` node is encountered the `evaluate_access` and `add_pattern` methods are invoked for each one of the statement's access patterns. Furthermore, the model extraction functions take a Layer-3 `pollyDomain` object as an argument simply because these functions are going to update (or access) the current state of the polyhedral domains (Section 3.3) represented by the `pollyDomain` input.

The default `RosePollyBuildModel` function uses a default implementation of the `RosePollyCustom` interface but the user can obviously customize the evaluation and modeling procedures by implementing a new subclass or extend the default one. In this case the `RosePollyBuildModel` class is overloaded to a new version that takes an additional `RosePollyCustom` argument as shown in Figure 5.5.

Notice that dependence analysis is done within `RosePollyBuildModel` too and uses a visitor-pattern traversal over the `FlowGraph` as well. The only difference though is that the `FlowGraph` now contains `affineStatement` nodes as opposed to the generic ones we used to construct the `FlowGraph`. Evidently, the `affineStatement` nodes – a subclass of the generic `Statement` node – carry the polyhedral model of each statements execution space and access patterns which are used by the dependence analysis algorithm. Our dependence analysis algorithm was an adaptation of the one proposed by Griebl et al. [Gri04] which is a value-based algorithm. However, we noticed that dependences like the true dependence $S2 \rightarrow S3$ of Figure 5.6 cannot be eliminated. Eliminating such dependences requires an additional post-processing step based on Wonacott and Williams [PW94, PW95], that has not been examined by this thesis.

```
for ( i=0 ; i < n ; i++ ) {
    S1: A[i+1] = ...
    S2: A[i+2] = ...
    S3: ... = A[i]
}
```

Figure 5.6: The $S2 \rightarrow S3$ true dependence cannot be elimitated by our dependence analysis algorithm.

After obtaining valid `RosePollyModel` objects from the `RosePollyBuildModel` method the user can use them to instantiate objects of subclasses that implement a specific optimization or analysis policy. In other words, by inheriting the polyhedral model from the `RosePollyModel` base class, subclasses may implement their own optimization or code generation strategy which could be original, i.e. direct child of the `RosePollyModel` class, or an extension of an existing one. For example the `RosePluto` and `RoseCloog` classes implement the Pluto and CLooG algorithms for scheduling and code generation respectively. Both of these classes can be extended to support an improved version of their algorithms or an extension to support GPU-specific scheduling or code generation strategies.

Section 5.4 presents a usage example of Layer-1 objects. In this example we see that for every (annotated) SCoP found in the input program, a `RosePollyModel` object is constructed. Then, for each one of these objects two Pluto objects are constructed each one invoked with a different fusion option (a Pluto-specific option). We then convert the derived schedules back into syntactic form using a pair of `RoseCloog` objects to match the two Pluto ones per SCoP. By comparing some cost function implemented by the Cloog class we can decide which one of the two Pluto options is the best for our program. This example shows that RosePolly enables us to build custom compilation flows very easily in a programmable fashion.

5.4 Layer-1 usage example

In the following usage example the `SgProject` object represents the entire syntax-tree of a compilation unit while the `frontend` method embodies the ROSE EDG front-end. Evidently

the `SgProject` class is a subclass of `SgNode`.

```
int main( int argc, char * argv[] ) {
    vector<string> argvList(argv,argv+argc);
    SgProject * proj = frontend(argvList);

    vector<RosePollyModel*> scops = RosePollyBuildModel(proj,ANNOTATED);
    for ( int i = 0 ; i < scops.size() ; i++ ) {
        /* The pluto object gets a complete copy of the polyhedral model (but no schedule) */
        RosePluto * pluto = RosePollyBuildPluto( scops[i] );
        /* Application of the Pluto algorithm with the MAX_FUSE option */
        pluto->apply(MAX_FUSE);
        /* cloog1 gets a copy of the pluto object including the derived schedules */
        RoseCloog * cloog1 = RosePollyBuildCloog( pluto );
        /* A new application of the Pluto algorithm overwrites any existing schedules */
        pluto->apply(SMART_FUSE);
        /* cloog2 now gets the new schedules derived with the SMART_FUSE option */
        RoseCloog * cloog2 = RosePollyBuildCloog( pluto );
        CloogOptions * opts = RoseCloog::init_default_options();

        cloog1->apply(opts);
        cloog2->apply(opts);

        /* e.g. degrees of parallelism */
        int metric1 = cloog1->get_metric();
        int metric2 = cloog2->get_metric();

        FlowGraph * graph = (metric1>=metric2) ?
        cloog1->print_to_flow_graph() :
        cloog2->print_to_flow_graph();

        RoseCUDA * cuda = RosePollyBuildCUDA( graph );
        /* more steps ... */
        }
        RosePollyTerminate();
    }
```

5.5 Layer-2 – The Polyhedral Model Interface

As we see from the UML diagram of Figure 5.1, layer-2 encompasses the main attributes of the `RosePollyModel` class. More specifically we see that every `RosePollyModel` object consists of a set of `affineStatement` objects, a set of `affineDependence` objects and a symbol table that stores information about the data of the SCoP. Each `affineStatement` object has an execution domain (see Section 3.3) and a schedule (see Section 3.4) captured by the layer-3 `pollyDomain` and `pollyMap` classes respectively. In addition, each `affineStatement` has a set of `accessPattern` objects corresponding to the data accesses of the respective computation statement. On the other hand, `affineDependence` objects contain two `affineStatement` objects, corresponding to a source and a sink statement, as well as a `pollyMap` object that captures the dependence polyhedron of the dependence (see Section 3.6). The entries of the symbol table are `Datum` objects each one containing `accessPattern` objects referring to specific access functions found in the input program. Notice that the same `accessPattern` objects can be accessed both from a given `Datum` as well as a given `affineStatement`. With such design it is easy to reason about access functions from a global or a statement-wise perspective. Furthermore, `Datum` objects carry unified global information about specific data something that can be very useful in compiling for heterogeneous architectures like GPUs, where it is often necessary to transfer data to and from device memory before and after the execution of a SCoP. In particular, each `Datum` carries a flag that specifies whether the respective data object was globally written to, read from or both and also a `pollyMap` (layer-3 object) that represents the collective global footprint of the `Datum`.

5.6 Layer-3 – The Math Interface

The third and last layer is the one responsible for managing two fundamental concepts of the polyhedral model, i.e., the concept of a Z-Polyhedron and the concept of an affine mapping (or affine function). The former is captured by the `pollyDomain` class while the latter is captured by the `pollyMap` class. Both of these classes provide methods for easy manipulation of affine

constraints and utilization of important operations like lexicographic minimization (i.e. integer linear programming or ILP) and Fourier-Motzkin elimination. In the current implementation of RosePolly, the `pollyDomain` and `pollyMap` classes are wrapper classes around a third-party polyhedral library[9] that handles all the low-level implementation details (see Section 3.8). By adding this layer of abstraction, we can effectively decouple the logic and semantics of our compiler from low-level implementation details related to the mathematical background of the polyhedral model – something that is outside the scope of our work.

First of all, the `pollyDomain` class represents a Z-polyhedron or in other words a domain as we defined it in Section 3.3. Therefore, each `affineStatement` has a `pollyDomain` that captures the execution domain of the respective computation statement. On the other hand, the `pollyMap` class consists of an affine mapping (or affine function) and a domain in which the mapping is valid. `pollyMap` is used by all Layer-2 objects as wee see in Figure 5.1. For the `affineDependence` a `pollyMap` is used to capture the dependence polyhedron of the dependence as defined in Section 3.6. For the `affineStatement` it is used to capture the schedule of the statement (see Section 3.4) which could be the result of the Pluto scheduling algorithm. Finally, for access functions and data (see Section 3.5), `pollyMap` represents their access functions.

Perhaps the most important concern with respect to the `pollyMap` and `pollyDomain` classes is how we can efficiently manipulate the underlying systems of affine constraints while minimizing the users exposure to library-specific implementation details. In RosePolly we provide two main avenues for doing that as we see in Figure 5.7. The first avenue is based on a light-weight matrix abstraction called `simple_matrix` that is used to store constraint coefficients using the built-in `int` data type and a well-defined matrix layout. However, in some cases it might be necessary to handle rational coefficients or utilize a special feature found in some third-party library. For cases like these we provide the `integer_map` and `integer_set` macros that expand to library-specific data types. This feature enables the user to directly access and utilize the underlying third-party library effectively bypassing the Layer-3 `pollyMap` and `pollyDomain` abstractions.

[9]This library is currently the ISL library.

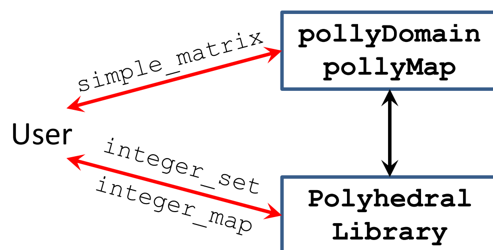


Figure 5.7: Layer-3 interface to the underlying systems of affine constraints.

5.7 Conclusions

In this chapter we presented the design and implementation of *RosePolly*, an object-oriented API for polyhedral compilation. To the best of our knowledge this is the first attempt to realize such framework. By taking advantage of inheritance and polymorphism, RosePolly enables the user to extend and customize existing polyhedral functionality very easily and therefore build polyhedral compilers and tools fast. This was proven extremely helpful for our primary research objectives as we will see in the next part of this Thesis. In particular, by using RosePolly we were able to build important GPU-specific tools fast, something that would otherwise be a rather slow and tedious process. Furthermore, we believe that this chapter will be very helpful for the reader too since it would be straight forward to associate the theoretical polyhedral concepts of Part II with the well-defined implementation design of RosePolly.

Part II

Code Generation

Chapter 6

Parametric GPU Code Generation for Static Control Programs

In Chapter 4, we saw how a static control program can be analyzed and automatically parallelized in the polyhedral model using the Pluto scheduling algorithm. That is, we identified and exposed parallelism at the finest granularity with respect to the SCoP execution spaces at hand (i.e. domains), which is a machine-independent characterization that ignores the finite nature of the underlying computing machine. Our task now is to partition parallelized SCoPs into independent chunks of finite size (i.e. extract coarse-grained parallelism) and schedule those chunks for execution on a GPU. In the context of OpenCL and CUDA this translates to the definition of a uniform rectangular partitioning of the parallel execution space – embodied by a work-group/nd-range configuration (Figure 2.6) – where each partition is subject to a fine-grained distribution of resources that has a direct yet hard to estimate impact on performance as we saw in Section 2.1.4.

This chapter presents and evaluates a code-generation scheme for producing parametrically partitioned static control programs for GPU execution. This scheme allows us to search for the right partitioning parameters at run-time and therefore avoid the cost of complex compile-time performance models or iterative compilation. The proposed mechanism is based on parametric tiling for producing parallel rectangular partitions of parametric size and a novel run-time

system that manages GPU execution and local memory usage dynamically. An experimental evaluation demonstrates the effectiveness of our approach for a variety of SCoPs from the *PolyBench* suite[1].

6.1 Introduction

One way to partition a SCoP for coarse-grained parallelism is by applying the well-known loop-tiling transformation[2] [WL91, LRW91, IT88] provided that it is semantics-preserving i.e. it respects the data-access dependences of the program. In the polyhedral model, loop-tiling can be effectively represented [BF03, LLL01, IT88, Gri04] as long as all tile sizes involved are compile-time literals and therefore preserve the affine characterization of the program. On the other hand, if tile sizes are parametric (unknown during compilation), we need to resort to non-polyhedral parametric tiling methods. Such methods have been proposed for perfectly nested loops [KRR+07, RKRS07] as well as arbitrary nested ones [KR, HBB+09].

However, the purpose of obtaining a tiled execution space (parametric or non-parametric) is typically to optimize a program for locality assuming that the data accessed by each tile can fit into higher levels of the memory hierarchy. Utilizing a tiling transformation for coarse-grained parallelism requires additional steps in order to identify tiles that can be executed in parallel. This can be exemplified by Figure 6.1 where even though the inner loop of the point execution space of Figure 6.1(a) is always parallel, the respective rectangularly-tiled space of Figure 6.1(b) has no parallel dimensions.

One of the fundamental properties of tileable loop nests, i.e., loop nests for which tiling is legal, is that they always allow parallel execution of tiles through wavefront/pipeline parallelism [Wol86, Wol89, Xue00]. For example in Figure 6.2 we see two tile-size configurations for a rectangularly-tiled execution space. Each dashed line represents a wavefront instance and all tiles that lie on the same dashed line (i.e. on the same wavefront instance) can always be executed in parallel.

[1] `www.cs.ucla.edu/~pouchet/software/polybench`

[2] An alternative method was proposed by Yang et al. [YXKZ10] based on *thread-merging* and *block-merging* transformations of CUDA kernels.

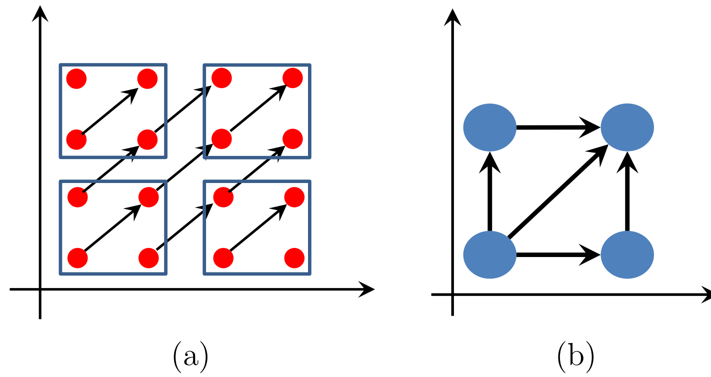


Figure 6.1: (a) A rectangularly tiled iteration space where the inner loop is always parallel, (b) The resultant tile-space where none of the tile dimensions are parallel. The large blue points in (b) correspond to the rectangular tiles of (a).

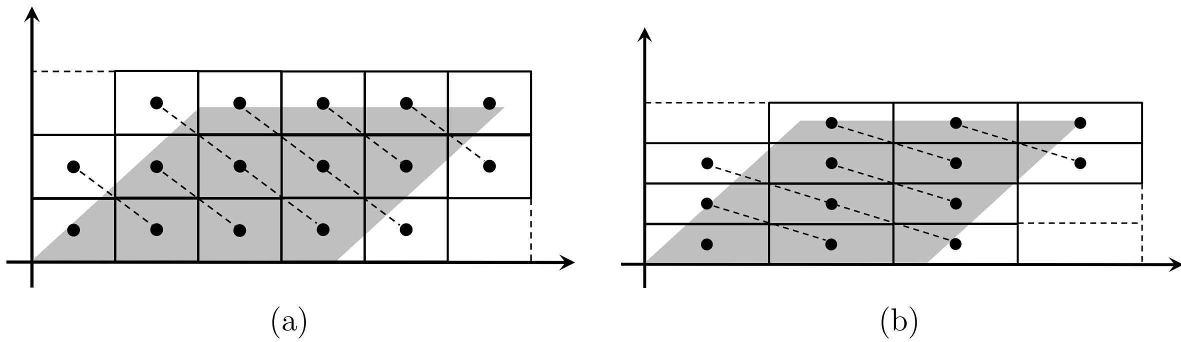


Figure 6.2: A rectangularly-tiled execution space using two different tile-size configurations. The grey area represents the actual execution space and the black rectangles represent the tiles.

The wavefront instances of course must be enumarated sequentially.

If tile-sizes are known during compilation then wavefronts of parallel tiles can be effectively generated using polyhedral code generation for SMP [BBK+08b, BR07] as well as distributed memory tragets [Gri04]. In the context of parametric tiling (also referred to as parameterized tiling), Hartono et al. [HBRS10, HBB+09] showed that a run-time system can be used to construct wavefronts of parallel rectangular tiles dynamically while Baskaran et al. [BHT+10] proposed a relaxed Fourier-Motzkin elimination algorithm in order to produce parameterized wavefronts of rectangular tiles at compile-time with no additional run-time support.

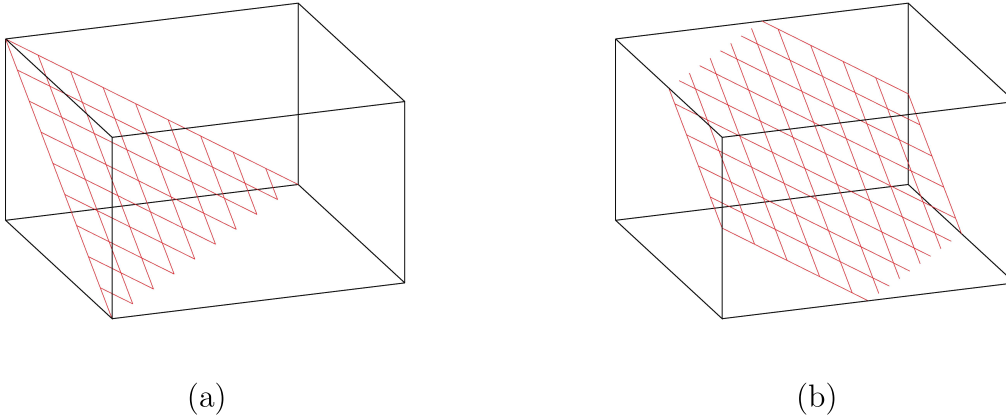
However, wavefront parallelism is not the only way of extracting parallel tiles. Krishnamoorthy et al. [KBB+07] and Strout et al. [SCF+05] proposed alternative methods that can be applied to simple stencil computations namely *split-tiling* and *overlapped-tiling*. The main weakness of

these methods though is their restrictive applicability which is why we do not examine them in more detail here.

With respect to GPU code generation, static partitioning (i.e., tile-sizes are known during compilation) and generation of wavefront parallelism has been the dominant strategy [BRS10, VCJC+13, ACE+12, DX11] with split-tiling [GCK+13] and overlapped-tiling [HPS12, MS09] schemes also being proposed. To the best of our knowledge there hasn't been any work that generates parametrically tiled GPU code for SCoPs to date, which constitutes the primary motivation for our work.

In Section 2.1.4 we saw that estimating the performance of GPU programs with reasonable accuracy, requires a detailed software and hardware analysis that is directly exposed to the partitioning of our program. This observation, combined with the diverse and evolving hardware organization of modern GPUs, highlights the importance of finding the right set of partitioning parameters for best performance through static (i.e. iterative compilation) or run-time tuning. The benefits of performing such tuning at run-time (as opposed to iterative compilation), is that we can minimize total compilation cost and enable fast design-space exploration across GPU devices by a single parameterized program. Parametric tiling can realize these benefits as it produces tiled loop nests with parametric tile sizes amenable to run-time tuning. However, in order to implement a parametric tiling scheme for GPUs we need to address the following technical challenges.

First, extracting and mapping parameterized wavefronts of parallel tiles for GPU execution can lead to load imbalance if wavefronts are not mapped precisely to a rectangular GPU execution environment. For example, in figure 6.3 we see a 3D execution space and the respective non-rectangular wavefronts. Executing those wavefronts on a GPU normally requires a rectangular over-approaximation that obviously leads to redundant allocations of resources both on the tile and intra-tile space. For the later, such load imbalance can have a severe impact on performance bacause it directly affects register and local memory utilization from redundant thread allocations. In addition, the parametric nature of the produced code indicates the need for a dynamic local memory management mechanism that would be able to allocate and use local



(a) (b)

Figure 6.3: Examples of non-rectangular wavefronts for a 3D execution space.

memory buffers dynamically. In this chapter we provide an answer to these technical challenges and formulate the first code generation algorithm for SCoPs that produces parameterized GPU code amenable to run-time tuning.

Our compilation flow (Figure 6.4) begins with a SCoP that enters a pre-processing stage embodied by an abstract polyhedral compilation framework (upper section of compilation flow) consisting of a model extraction, scheduling and syntax recovery module as shown in Figure 6.4 and explained in Section 3.7. This framework is used to find combinations of loop-nest transformations – in the form of affine scheduling functions – that enable tiling. For the remainder of this chapter we are assuming that this first pre-processing step produces a tilable SCoP through some automatic scheduling algorithm like Pluto (Chapter 4). This chapter is focused on the lower section of the compilation flow graph of Figure 6.4 where we first produce a parametric tile space and an intra-tile version of the input SCoP at compile-time (Section 6.2). We then map the produced tile space to a GPU execution environment at run-time and use the intra-tile version of our SCoP to produce the GPU device code (Section 6.3). Finally, we show how we manage local memory usage dynamically by a combination of compile-time and run-time methods (Section 6.4).

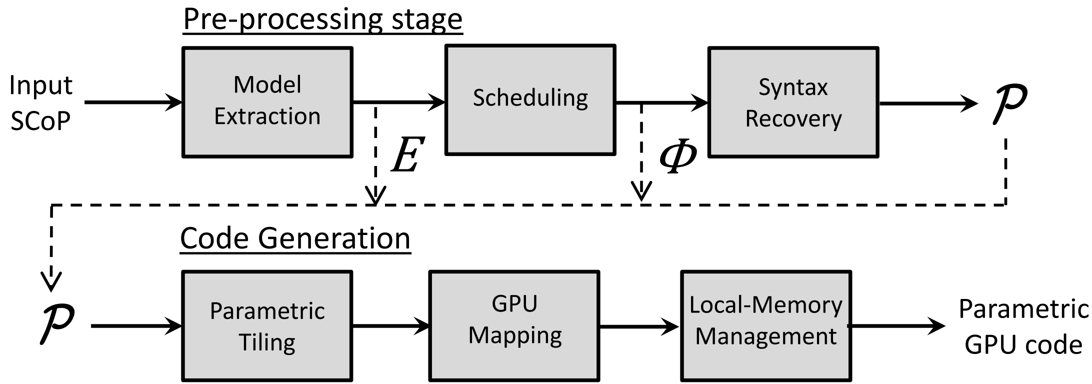


Figure 6.4: Code generation flow, where E is a set of polyhedral dependences, Φ a set of multi-dimensional affine transformations and $\mathcal{P}$ a syntactic form of a tileable program.

6.2 Parametric Tiling

The proposed code-generation scheme relies on two independent pre-processing steps corresponding to Sections 6.2.1 and 6.2.2. The first step (Section 6.2.1) utilizes well-known techniques [BHT+10] for determining a parametric tile space for the input program in the form of perfectly nested loops that scan a uniform space of rectangular tiles with parametric sizes like the ones shown in Figure 6.2.

The second step (Section 6.2.2) focuses on the intra-tile space, i.e., on the rectangular execution space enclosed within each tile. The objective of both steps is to expose coarse-grained and fine-grained parallelism respectively either through wavefronts or through rectangularly parallel loop dimensions if any (the trivial case).

6.2.1 The Tile Space

First of all, in order to ensure that tiling is a semantics preserving transformation, we define the legality condition for tiling based on Section 3.6 definition of dependence vectors :

Definition 3. *For a d-dimensional problem, tiling is a legal transformation iff* $\delta_{e_i} \geq 0$ *for all dimensions* $i \in [1 \ldots d]$ *and for all dependences* $e \in E$*.*

Definition 3 essentially states that tiling is not legal if there exists a dependence edge $e \in E$ that yields a non-lexicographically positive dependence vector [WL91].

Tiling is usually enabled by a loop skewing transformation [Wol86] however, deriving the right combination of loop transformations to enable tiling in the general case of arbitrary SCoPs can be effectively achieved through polyhedral scheduling (see Chapters 3 and 4). Consequently, let $\Phi : \{\phi_{S_1} \dots \phi_{S_n}\}$ be a set of multi-dimensional affine transformations (i.e. schedules as per Section 3.4) derived by a polyhedral scheduling algorithm (e.g. Pluto). Dependence vectors are now defined for a transformed d-dimensional program as follows :

$$e_{tv}(e) = (\delta_{e_{t0}}, \dots, \delta_{e_{td}}),$$

$$\delta_{e_{ti}} = \begin{cases} 1 & \text{if } \phi_{sink_e}[i](\vec{x}_{sink_e}) - \phi_{src_e}[i](h_{e_i}(\vec{x}_{sink_e})) > 0, \\ 0 & \text{if } \phi_{sink_e}[i](\vec{x}_{sink_e}) - \phi_{src_e}[i](h_{e_i}(\vec{x}_{sink_e})) = 0, \\ -1 & \text{if } \phi_{sink_e}[i](\vec{x}_{sink_e}) - \phi_{src_e}[i](h_{e_i}(\vec{x}_{sink_e})) < 0. \end{cases}$$

$$\text{for } i \in [0..d]$$

In case there is still a dependence edge $e \in E$ with a negative component for $e_{tv}(e)$, we can attempt to find a set of innermost schedule dimensions, for which Definition 3 is satisfied. Therefore, let $d_t \leq d$ denote the innermost dimensions of ϕ_{S_i} for which tiling is legal across all statements S_i.

Given d_t innermost tileable dimensions, we define $L_i : i \in [1 \dots d_t]$ to be a set of perfectly nested loops that scan a space of uniform rectangular tiles of parametric sizes. Each loop L_i will effectively represent a coordinate dimension and each tile will be uniquely identified by an iteration vector $\vec{t} = [t_1 \in L_1, \dots, t_{d_t} \in L_{d_t}]$. Given a set L_i of tile loops, we can seek a subset of rectangularly parallel tile loops $L_{P_i} : i \in [1 \dots d_{par} \leq d_t]$ that we can map directly into an ND-range or resort to wavefront parallelism if $d_{par} = 0$. In the former case, let $\mathbb{S}_T$ and $\mathbb{S}_{TP}$ denote the sets of tile loops L_i and rectangularly parallel tile loops L_{P_i} respectively.

In the general case of arbitrary SCoPs we define $\mathbb{S}_T$ as the set of tile loops needed to scan the tiled convex hull of the d_t innermost dimensions of the transformed execution space. We begin by recovering the syntax of the input SCoP under Φ, thus getting a new syntactic tree $\mathcal{P}$.

Afterwards, we acquire the new transformed domains D'_{S_i} for each statement in $\mathcal{P}$ and use a polyhedral library (Section 3.8) to recover the convex hull D_{CH} of the d_t innermost dimensions of all D'_{S_i}. In case of imperfectly nested programs we add semantics preserving one-time-loops (OTLs) on $\mathcal{P}$ prior to extracting D'_{S_i} (see Figure 6.5 for an example of OTLs). Note that D_{CH} is a convex polyhedron by definition, thus represented by a single integer coefficient matrix.

Instead of transforming D_{CH} into a syntactic form, we use a more appropriate data structure that facilitates robust algebraic manipulation of non-affine loop-bounds. In particular, each row of D_{CH} is turned into a loop bound expression implemented as a list of symbolic polynomial fractions as suggested by Baskaran et al. [BHT+10]. Let lb_i and ub_i denote the lower and upper bound expressions respectively, for loop $L_i \in \mathbb{S}_T$ with $i \in [1 \ldots d_t]$. We now have a fully permutable vector of polynomial expression pairs (each pair consisting of a lower and upper bound expression) that we use to apply the following algebraic operations to get the final tiled execution space $\mathbb{S}_T$:

Introduce tile coordinates Each coordinate x_i of the original execution space D_{CH}, is expressed in terms of tile coordinates t_i, intra-tile coordinates u_i and tile sizes T_i as follows :

$$x_i = t_i \cdot T_i + u_i, \qquad \text{for } 0 \leq u_i < T_i$$

Finally, $\mathbb{S}_T$ takes the form :

$$\mathbb{S}_T : lb_i \leq t_i \cdot T_i + u_i \leq ub_i, \qquad \text{for } i \in [1..d_t]$$

Eliminate intra-tile coordinates The intra-tile coordinates u_i can be eliminated by making sure we include all non-empty tiles :

$$\mathbb{S}_T : \begin{pmatrix} lb_i \leq t_i \cdot T_i + T_i - 1 \\ ub_i \geq t_i \cdot T_i \end{pmatrix}, \qquad \text{for } i \in [1..d_t]$$

Get final tile loop-bounds The resulting expressions require additional processing since the tile coordinate variables appear as part of a product (i.e. $t_i \cdot T_i$) that prevents us from

constructing tile loops. We overcome this by dividing all terms by T_i and since tile coordinates can only take integer values, we enclose the resulting symbolic fractions into *floor*[3] operators :

$$\mathbb{S}_T : \begin{pmatrix} floor(lb_i/T_i) + floor(1/T_i) - 1 \leq t_i \\ floor(ub_i/T_i) \geq t_i \end{pmatrix}, \qquad \text{for } i \in [1..d_t]$$

The introduction of *floor* operations has an important impact on the resulting tile loops. In particular, they produce a number of empty tiles i.e. tiles that do not include any valid points. In Section 6.3 we'll show how empty tiles are eliminated simply by using the expressions of the previous step as run-time conditional predicates.

From Definition 3 we know that all d_t innermost dimensions of dependence vectors $e_{tv}(e)$ are either 0 or 1. A value of 1 indicates that there is an inter-tile dependence across the corresponding tile dimension/loop. On the other hand, a tile loop $L_i \in \mathbb{S}_T$ is parallel and can be mapped into an ND-Range if the following condition holds :

Definition 4. *A tile loop $L_i \in \mathbb{S}_T$ with $i \in [1..d_t]$ is parallel iff $\delta_{e_{ti}} = 0$ for all $e \in E$. Otherwise, it is sequential.*

Let $\mathbb{S}_{TP}$ be the set of loops that satisfy Definition 4 with $Card(\mathbb{S}_{TP}) = d_{par}$. If this set is non-empty, i.e. $d_{par} > 0$, then the loops $L_{Pi} \in \mathbb{S}_{TP}$ can be mapped directly into an ND-range while the remaining loops $L_{Si} \in \mathbb{S}_T \wedge L_{Si} \notin \mathbb{S}_{TP}$, if any, will be pushed into the device code and executed sequentially by each thread. Otherwise, if no parallel tile loops can be found, we resort to wavefront parallelism.

In theory, a wavefront can be modeled by a hyperplane of the tile space defined as :

$$\mathcal{W}(\vec{t}) = I^{\top} \cdot \vec{t}, \qquad \text{for } \vec{t} \in \mathbb{S}_T \tag{6.1}$$

This hyperplane guarantees that all tile instances that belong to the same wavefront instance are independent and thus can be executed in parallel. Formally :

[3] A *floor* operator returns the largest integer that is not greater than the actual result of the fraction.

Definition 5. *Let $\vec{t}$ and $\vec{t'}$ be two valid tile instances. If $\mathcal{W}(\vec{t}) = \mathcal{W}(\vec{t'})$, then $\vec{t}$ and $\vec{t'}$ are independent and can be safely executed in parallel.*

Proof. If we ignore the obvious case where $\vec{t} = \vec{t'}$, then according to Equation (6.1) in order for $\mathcal{W}(\vec{t}) = \mathcal{W}(\vec{t'})$ to hold we have :

$$t_1 + t_2 + \cdots + t_{d_t} = t'_1 + t'_2 + \cdots + t'_{d_t} \tag{6.2}$$

By rearranging the terms of (6.2) we have :

$$\underbrace{(t_1 - t'_1)}_{\Delta_1} + \underbrace{(t_2 - t'_2)}_{\Delta_2} + \cdots + \underbrace{(t_{d_t} - t'_{d_t})}_{\Delta_{d_t}} = 0 \tag{6.3}$$

Since $\vec{t} \neq \vec{t'}$ then $\exists i \in [1..d_t]$ s.t. $\Delta_i \neq 0$. If $\Delta_i > 0$ then in order for (6.3) to hold there must be an $i' \in [1..d_t]$ s.t. $\Delta_{i'} < 0$. Therefore, if there exists a dependence $e \in E$ that involves $\vec{t}$ and $\vec{t'}$ then $\delta_{e_{ti'}} = -1$ which contradicts Definition 3. Consequently, $\nexists e \in E$ that involves $\vec{t}$ and $\vec{t'}$ thus they can be safely executed in parallel.

□

In practice, if we combine (6.1) with system $\mathbb{S}_T$ of tile loops, and use Fourier-Motzkin elimination to eliminate $\vec{t}$ from (6.1) and $[t_{i+1} \dots t_{d_t}]$[4] from $L_i \in \mathbb{S}_T$ we could end-up with a wavefront system of loops. However, the tile bounds of the L_i loops involve parametric fractions of indeterminate sign which makes the classic Fourier-Motzkin elimination algorithm inapplicable. In order to overcome this problem we utilize the relaxed Fourier-Motzkin elimination algorithm proposed by Baskaran et al. [BHT+10] and produce the desired system of wavefront loops :

[4] Innermost tile coordinates.

$$\mathcal{W} : lbw \leq w \leq ubw$$

$$\mathbb{S}_{WT}(w) : lb_{wi}(w) \leq t_i \leq ub_{wi}(w), \quad \text{for } i \in [1..d_t]$$

Evidently, loops $\mathbb{S}_{WT}(w)$ from the above system enumarate the tiles within each wavefront instance w and can be executed in parallel while the wavefront loop $\mathcal{W}$ enumarates the wavefront instances and is executed sequentially.

Algorithm

Input : A set of multi-dimensional schedules $\Phi : \{\phi_{S_1} \ldots \phi_{S_n}\}$. A set of polyhedral dependences E and a syntactic form of the program $\mathcal{P}$ that was recovered under schedules Φ with a syntax-recovery tool.

Output : Sets $\mathbb{S}_T$ and $\mathbb{S}_{TP}$ or sets $\mathcal{W}$ and $\mathbb{S}_{WT}(w)$ with $w \in \mathcal{W}$, depending on whether we resorted to wavefront parallelism or not. All the returned sets are vectors of symbolic expressions implemented according to [BHT+10].

Step 1 For each dependence edge $e \in E$ get dependence vector $e_{tv}(e)$ using schedules Φ.

Step 2 Based on dependence vectors $e_{tv}(e)$ and Definition 3 determine the innermost tileable dimensions of Φ (usually all of them). This step returns d_t which denotes the number of innermost tileable dimensions.

Step 3 Based on dependence vectors $e_{tv}(e)$, determine which tileable dimensions are parallel according to Definition 4. For this step, a bit vector $\vec{p}$ of size d_t is used in order to flag the parallel dimensions. The total amount of parallel dimensions is denoted by d_{par}.

Step 4 In case of imperfectly nested loops add semantics preserving one-time loops (OTLs) to $\mathcal{P}$ and get a new syntactic tree $\mathcal{P}'$ in which all syntactic statement instances are surrounded by the same number of loops (see Figure 6.5 for an example).

Step 5 Acquire the new transformed domains D'_{S_i} for each statement S_i in $\mathcal{P}'$.

Step 6 Get the convex hull D_{CH} of the d_t innermost dimensions of all D'_{S_i}. D_{CH} is represented by a single integer coefficient matrix.

Step 7 Convert D_{CH} into a vector $\mathbb{S}_T$ of symbolic polynomial expressions implemented according to [BHT+10].

Step 8 If $d_{par} = 0$ go to **Step 8.1**. Otherwise go to **Step 8.2**.

Step 8.1 Invoke RSFME algorithm [BHT+10] and get $\mathcal{W}$ and $\mathbb{S}_{WT}(w)$ from $\mathbb{S}_T$. Go to **Step 9**.

Step 8.2 Remove all parallel tile dimensions from $\mathbb{S}_T$ using $\vec{p}$, and place into new vector $\mathbb{S}_{TP}$ of parallel tile dimensions. Go to **Step 9**.

Step 9 If $d_{par} = 0$ return $\mathcal{W}$ and $\mathbb{S}_{WT}(w)$. Otherwise, return $\mathbb{S}_T$ and $\mathbb{S}_{TP}$.

Implementation

Evidently, Algorithm 6.2.1 can be implemented with RosePolly (Chapter 5), even though an existing implementation provided by PoCC was actually used for our experiments. In particular, we can use method `RosePollyBuildModel` to get a `RosePollyModel` object from our input SCoP effectively extracting the polyhedral model including the dependences E. Then we can instantiate a `RosePluto` object and a `RoseCloog` object exactly like we see in the Layer-1 usage Example of Section 5.4. After invoking the Pluto algorithm using `RosePluto::apply` we get the schedules Φ as the `pollyMap` members of each `affineStatement` of `RosePluto`. Then we can invoke `RoseCloog::apply` and `RoseCloog::print_to_flow_graph` to get the transformed program $\mathcal{P}$. This means that `RoseCloog` can give us $\mathcal{P}$ in the form of a `FlowGraph` object and therefore, adding the OTLs in step 4 can be as easy as adding `ForLoop` nodes in the $\mathcal{P}$ and get $\mathcal{P}'$ as a new `FlowGraph`.

Acquiring the new transformed domains D'_{S_i} for step 5 can be done by invoking the `RosePollyBuildModel` method with `FlowGraph` $\mathcal{P}'$ as an input (the inner grey box of Figures 5.3 and 5.5). Afterwards,

the convex hull computation of step 6 can be implemented as a `RosePollyModel::convex_hull` method that invokes a Layer-3 `pollyDomain::convex_hull` operation incrementally for pairs of `affineStatement` objects. The result would be a Layer-3 `pollyDomain` object that represents the convex domain. Finally we can request a `simple_matrix` object from the convex domain that we can then use to construct our symbolic polynomial expressions for step 7.

6.2.2 The intra-Tile Space

The situation within each tile appears to be simpler as it is just a rectangular execution space. Nevertheless, in order to preserve the legality of tiling we need to respect the multi-dimensional schedules Φ embodied by the transformed syntax $\mathcal{P}'$. Furthermore, we need to identify parallelism within each tile as well, which might come from parallel intra-tile dimensions or wavefront parallelism.

In either case, parallel intra-tile points will be captured by a work-group configuration and executed by the device code in a SIMT fashion. Since the respective work-group configuration will inherently respect the tile bounds of the parallel intra-tile dimensions, we only need to replace the respective syntactic loop bounds of $\mathcal{P}'$ with if-guards and adjust non-parallel loop bounds to be :

$$\mathbb{S}_{I_{seq}} : max(lb_i, t_i \cdot T_i) \leq x_i \leq min(ub_i, t_i \cdot T_i + T_i - 1) \tag{6.4}$$

The result is a transformed syntax tree $\mathcal{P}_{intra}$ that will be used to produce the device code (Section 6.5).

In the case of wavefront parallelism the situation is rather straightforward. In particular, since the intra-tile space is essentially a rectangular bounding box with T_i extents across each dimension $i \in [1..d_t]$ the wavefront loops can be generated for the intra-tile space with hyperplane (6.5) using a polyhedral code-generation tool [Bas04, QRW00, Che12]. We can then wrap these loops around $\mathcal{P}'$ and replace all but the outer wavefront loop with if-guards. Figure 6.5 shows how

this process can be applied to the ADI benchmark from the polybench suite[5]. In Section 6.3 we'll see that the wavefront conditions can actually be hoisted to the host code and evaluated once using an automatic run-time mechanism. Therefore, $\mathcal{P}_{intra}$ in case of intra-tile wavefront parallelism, is actually produced by replacing all loops in $\mathcal{P}'$ with if-guards (Figure 6.5(b)).

$$\mathcal{W}_I(\vec{u}) = I^\top \cdot \vec{u}, \qquad \text{for } 0 \leq u_i < T_i \qquad \text{and } i \in [1..d_t] \tag{6.5}$$

Sometimes, the derived schedules $\phi_{S_i} \in \Phi$ for an input SCoP, will result in a maximally fused target loop-nest in an attempt to minimize sequential execution overhead. However, in a GPU execution context this approach is not always ideal as we see from the Jacobi-2d example of Figure 6.6. In particular, we notice that the schedules derived from the Pluto scheduling algorithm (Chapter 4), resulted in a maximally fused program, where inter-statement dependences carried by the space dimensions $\phi_{S_i}[1]$ and $\phi_{S_i}[2]$, prevent the respective space loops (i.e. loops i and j) from being parallel. This situation forces us to resort to wavefront parallelism on the intra-tile space as well. However, avoiding an intra-tile wavefront – if possible – can be highly beneficial because the lightweight nature of GPU cores, makes them particularly vulnerable to the additional control overhead incurred by wavefront parallelism.

In order to overcome this problem, we propose Algorithm 4 which is applied on Φ prior to acquiring $\mathcal{P}$, in order to eliminate such inter-statement dependences. More specifically, Algorithm 4 utilizes a directed dependence graph – where the vertices of the graph are the statements of the input SCoP – in order to extract strongly connected components. The strongly connected components are then decoupled by inserting scalar dimensions to all $\phi_{S_i} \in \Phi$ corresponding to the respective component identification number $scc[i]$ of each statement – a process similar to classic loop vectorization algorithms [AK87]. Figure 6.6 (c) shows the result for the Jacobi-2d example. Note, that Algorithm 4 does not alter the affine transformations per se, but only the fusion structure of the program in an attempt to avoid wavefront parallelism.

An important observation from Algorithm 4 is the following: if the condition of line 5 is false,

[5] www.cs.ucla.edu/~pouchet/software/polybench

```
for ( c0 = 0 ; c0 <= T-1 ; c0++ ) {
   for ( c1 = c0 ; c1 <= c0 ; c1++ ) { // OTL
      for ( c2 = c0+1 ; c2 <= c0+N-1 ; c2++ ) {
         S1; S2;
      }
   }
   for ( c1 = c0+1 ; c1 <= c0+N-1 ; c1++ ) {
      for ( c2 = c0+1 ; c2 <= c0+N-1 ; c2++ ) {
         S3; S4; S5; S6;
      }
      for ( c2 = c0+N ; c2 <= c0+N ; c2++ ) { //OTL
         S7; S8;
      }
   }
}
```

(a)

```
if ( c0 >= 0 && c0 <= T-1 ) {
   if ( c1 >= c0 && c1 <= c0 ) { // OTL
      if ( c2 >= c0+1 && c2 <= c0+N-1 ) {
         S1; S2;
      }
   }
   if ( c1 >= c0+1 && c1 <= c0+N-1 ) {
      if ( c2 >= c0+1 && c2 <= c0+N-1 ) {
         S3; S4; S5; S6;
      }
      if ( c2 >= c0+N && c2 <= c0+N ) { //OTL
         S7; S8;
      }
   }
}
```

(b)

```
for ( w = 0 ; w <= T0+T1+T2-3 ; w++ ) {

   // Wavefront conditions
   if ( c0 >= 0 && c0 >= w-T1-T2+2 && c0 <= w && c0 <= T0-1 ) {
      if ( c1 >= 0 && c1 >= w-c0-T2+1 && c1 <= T1-1 && c1 <= w-c0 ) {

         // Recover global coordinates
         c0 += t0; c1 += t1;
         c2 = t2 + (w-c0-c1);

         // Compute kernel
         if ( c0 >= 0 && c0 <= T-1 ) {
            if ( c1 >= c0 && c1 <= c0 ) { // OTL
               if ( c2 >= c0+1 && c2 <= c0+N-1 ) {
                  S1; S2;
               }
            }
            if ( c1 >= c0+1 && c1 <= c0+N-1 ) {
               if ( c2 >= c0+1 && c2 <= c0+N-1 ) {
                  S3; S4; S5; S6;
               }
               if ( c2 >= c0+N && c2 <= c0+N ) { //OTL
                  S7; S8;
               }
            }
         }
      }
   }
   // Synchronize
}
```

(c)

Figure 6.5: (a) Recovered syntax under pluto scheduling amended with OTLs (i.e. $\mathcal{P}'$), (b) All for-loops are replaced with if-guards (i.e. $\mathcal{P}_{intra}$) (c) The sequential wavefront loop and parallel wavefront conditions are wrapped around $\mathcal{P}_{intra}$ to produce the final ADI syntax.

```
for( t=0; t<T; t++ ) {
   for( i=2; i<N-1; i++ )
      for( j=2; j<N-1; j++ )
         S1(t,i,j);
   for( i=2; i<N-1; i++ )
      for( j=2; j<N-1; j++ )
         S2(t,i,j);
}
```

(a)

```
for( t=0; i<T; t++ ) {
   for( j=2t+2; j<2t+N-1; j++ )
      S1(t,2,j-2t);
   for( i=2t+3; i<2t+N-1; i++ ) {
      S1(t,i-2t,2);
      for( j=2t+3; j<2t+N-1; j++ ) {
         S2(t,i-2t-1,j-2t-1);
         S1(t,i-2t,j-2t);
      }
      S2(t,i-2t-1,N-2);
   }
   for( j=2t+3; j<2t+N; j++ )
      S2(t,N-2,j-2t-1);
}
```

$$\phi_{S_1} : (t, 2t+i, 2t+j)$$
$$\phi_{S_2} : (t, 2t+i+1, 2t+j+1)$$

(b)

```
for( t=0; t<N; t++ ) {
   for( i=2t+2; i<2t+N-2; i++ )
      for( j=2t+2; j<2t+N-2; j++ )
         S1(t,i-2t,j-2t);
   for( i=2t+3; i<2t+N-1; i++ )
      for( j=2t+3; j<2t+N-1; j++ )
         S2(t,i-2t-1,j-2t-1);
}
```

$$\phi'_{S_1} : (t, 0, 2t+i, 2t+j)$$
$$\phi'_{S_2} : (t, 1, 2t+i+1, 2t+j+1)$$

(c)

$$S1(x_1, x_2, x_3) : b[x_2][x_3] = 0.2 \cdot (a[x_2][x_3] + a[x_2][x_3-1] + a[x_2][x_3+1] + a[x_2+1][x_3] + a[x_2-1][x_3])$$
$$S2(x_1, x_2, x_3) : a[x_2][x_3] = b[x_2][x_3]$$

Figure 6.6: (a) The original Jacobi-2d kernel, (b) Transformed Jacobi-2d kernel using the Pluto scheduling algorithm [BBK+08b], (c) Proposed fusion structure derived from Algorithm 4.

then the rest of the scheduling dimensions are unfused and marked parallel. Therefore, in order to ensure correctness of the respective parallel program, a decoupling of strongly connected components imposed by line 6, must be accompanied by intra-tile synchronization in between those components. This is exemplified by the final version of $\mathcal{P}_{intra}$ for the Jacobi-2d kernel shown in Figure 6.7.

Algorithm

Input : A set of multi-dimensional schedules $\Phi : \{\phi_{S_1} \ldots \phi_{S_n}\}$ and a set of polyhedral dependences E.

Output : A syntactic tree $\mathcal{P}_{intra}$ representing the device code for our scop.

Step 1 Use E to construct a directed dependence graph *ddg* where each node is a statement and each edge represents a dependence $e \in E$.

Step 2 Use *ddg* from **Step 1** along with E and Φ as input to Algorithm 4. Get a set of new schedules Φ' in return along with vector *mark* specifying parallel intra-tile dimensions.

Algorithm 4 Elimination of intra-tile dependences that can result in unnecessary intra-tile wavefront. Let *ddg* be the directed dependence graph of a *d*-dimensional program derived from a set of polyhedral dependence edges $e \in E$ each involving a source (src_e) and a sink ($sink_e$) statement.

1: **procedure** INTRADEPELIMINATION(Φ,ddg,E)
2: $scc[1 \ldots n] \leftarrow ddg$ ▷ Calculate scc with well-known algorithms
3: $mark[1 \ldots d] \leftarrow parallel$ ▷ All loops marked parallel
4: **for each** $i \in [1..d]$ **do**
5: **if** ($\nexists e \in E$ **for which** $scc[src_e] = scc[sink_e]$) **then**
6: CUTSCC(i,scc) ▷ Add *scc* values to Φ on position *i*
7: **return** *mark*
8: **end if**
9: **if** (ISPARALLEL(i,Φ)=**false**) **then**
10: $mark[i]$ = non-parallel
11: update *E*,*ddg* and *scc* ▷ Remove satisfied dependences
12: **if** ($E = \oslash$) **return** *mark* ▷ Exit if no dependences left
13: **end if**
14: **end for**
15: **return** *mark*
16: **end procedure**

```
for( t=max(0,t1*T1); t<min(N,t1*T1+T1-1); t++ ) {

    if( i>=2t+2 && i<2t+N-1 )
        if( j>=2t+2 && j<2t+N-1 )
            S1(t,i-2t,j-2t);

    // Synchronization

    if( i>=2t+2 && i<2t+N-1 )
        if( j>=2t+2 && j<2t+N-1 )
            S2(t,i-2t,j-2t);

    // Synchronization
}
```

Figure 6.7: Final version of $\mathcal{P}_{intra}$ for the Jacobi-2d kernel. The time loop was marked sequential thus modified accordingly, while the parallel space loops were turned into if-guards.

Step 3 Use a syntax recovery tool to recover the syntax under Φ' and get syntactic tree $\mathcal{P}$.

Step 4 Invoke Algorithm 6.2.1 using $\mathcal{P}$ and get $\mathcal{P}'$ amended with OTLs in return.

Step 5 If *mark* indicates the existence of parallel intra-tile dimensions go to **Step 5.1**. Otherwise go to **Step 5.2**.

Step 5.1 Based on syntactic pattern-matching, locate the for-loops in $\mathcal{P}'$ that correspond to the parallel intra-tile dimensions indicated by *mark* and replace them with if-guards. Modify the remaining for-loops according to (6.4). The result would be $\mathcal{P}_{intra}$. Go to **Step 6**.

Step 5.2 Replace all for-loops with if-guards and get $\mathcal{P}_{intra}$ as a result. Go to **Step 6**.

Step 6 Return $\mathcal{P}_{intra}$

Implementation

First of all, steps 1 and 2 including Algorithm 4 can be implemented as private `RosePluto` methods that are invoked as part of the main `RosePluto::apply` method or as independent public methods of the `RosePluto` class. For step 3 we simply instantiate and use a `RoseCloog` object just like we did for Algorithm 6.2.1 and showed in the Layer-1 usage example of Section 5.4. Finally, because $\mathcal{P}'$ is in the form of a `FlowGraph` object returned by the `RoseCloog::print_to_flow_graph` method, steps 5.1 and 5.2 can be implemented as visitor-pattern traversals over $\mathcal{P}'$ that simply replace `ForLoop` nodes with `Conditional` nodes whenever necessary.

6.3 GPU Mapping

The GPU mapping process involves the task of mapping parallel tiles and parallel intra-tile points into the virtual processor space of an OpenCL device, embodied by the ND-Range and Work-Group configurations. We already mentioned in the introduction of this chapter that the

GPU mapping process is going to take place at run-time, i.e. it will rely on a run-time system that is inherently decoupled from the compiler. Two fundamental concepts in this run-time system are the *Tile-Bucket* and the *Thread-Bucket* defined as follows:

Definition 6. *A Tile-Bucket is denoted by $\mathcal{B}_T$ and contains the coordinates of all parallel tile instances to be mapped into an ND-Range and it is of size $\mathcal{B}_{Tsize}(w) = |\mathbb{S}_{WT}(w)|$ or $\mathcal{B}_{Tsize} = |\mathbb{S}_{TP}|$ for wavefront or rectangularly parallel tile spaces respectively.*

Definition 7. *A Thread-Bucket is denoted by $\mathcal{B}_I$ and contains the coordinates of all parallel intra-tile points to be executed by each work-group.*

Each bucket is populated dynamically at run-time by the host and then transferred into concurrent (i.e. shared) data structures residing in a thread-visible memory level (e.g. global memory) where each bucket entry (i.e. tile or intra-tile coordinate) can be recovered from the device code using the built-in index variables, i.e., **glw** and **gli** as we defined them in Section 2.1.3.

With respect to the tile-bucket, in Section 6.2.1 we defined our parallelized tile space as a vector of loop-bound expressions derived from rectangularly-parallel tile dimension – $\mathbb{S}_{TP}$ – or wavefront parallelism – $\mathbb{S}_{TW}(w)$. These loops can now be executed in any order from the host environment and populate the tile-bucket $\mathcal{B}_T$ with tile coordinates according to Algorithm 5. Note that the chosen execution order will effectively define the layout of the mapping. This layout could be an arbitrary permutation of the parallel tile loops or a more complex layout like a diagonal reordering to avoid partition camping [RM09]. In Chapter 7 we will show that defining the mapping layout at the host – rather than the device – has surprising benefits in programmability and debugging with negligible run-time overhead.

Algorithm 5 also includes two main optimizations i.e. empty tile elimination and full tile separation. In line 2 we use the intra-tile coordinate elimination conditions from Section 6.2.1 to eliminate all empty tiles resulted from the rounded parametric fractions used to generate the tile loop bound expressions. On the other hand, in lines 4-8 we use the following conditions in order to perform full tile separation :

Algorithm 5 Population of tile-bucket $\mathcal{B}_T$ with full-tile separation and empty-tile elimination.

```
 1: procedure PopulateTileBucket(t⃗,B_T,B_Tsize)                    ▷ Host Code
 2:     if (t⃗ is not an empty tile) then                   ▷ Emptiness conditions
 3:         B_T[B_Tsize] ← {t_1, ..., t_d_par}
 4:         if (t⃗ is not a full tile) then                  ▷ Fullness conditions
 5:             B_T[B_Tsize] ← is partial
 6:         else
 7:             B_T[B_Tsize] ← is full
 8:         end if
 9:         B_Tsize ← B_Tsize + 1
10:     end if
11:     return B_T, B_Tsize
12: end procedure
```

$$\bigvee_{i=1}^{d_{par}} ((t_i \cdot T_i < lb_i) \vee (ub_i < t_i \cdot T_i + T_i - 1)) \tag{6.6}$$

Notice that condition (6.6) might only be partially fulfilled by a given tile. However, at this stage we only distinguish between complete and no fulfillment of (6.6) and thus consider two versions of $\mathcal{P}_{intra}$ (see Algorithm 6.2.2), one representing partial tiles ($\mathcal{P}_{intra}$ without modifications) and one representing full tiles ($\mathcal{P}_{intra}^{full}$ where all if-guards are removed except for those corresponding to OTLs).

On the intra-tile level, if the number of parallel transformation dimensions $d_{par} \leq d_t$ is non-zero (see $mark$ vector of Algorithm 4), then we have a d_{par}-dimensional rectangle containing parallel execution instances that can be mapped directly into a Work-Group (without the use of thread-buckets).

In case of intra-tile wavefront parallelism, bucket $\mathcal{B}_I$ is split into multiple buckets, each one corresponding to a wavefront instance $w \in \mathcal{W}_I$ and containing the parallel execution instances of w. Therefore, $\mathcal{B}_I$ is defined as $\mathcal{B}_I[\mathcal{W}_{Isize}][\mathcal{B}_{Isize}]$ with $\mathcal{W}_{Isize}$ and $\mathcal{B}_{Isize}$ being defined as the maximum number of wavefronts and the maximum number of points within a wavefront respectively. Both $\mathcal{W}_{Isize}$ and $\mathcal{B}_{Isize}$ are symbolic expressions depending on tile sizes and are empirically hard-coded into the run-time system for 2D and 3D tiles.

Notice that $\mathcal{B}_{Isize}$ reflects the maximum number of intra-wavefront points across all wavefront instances. We will use it for the Work-Group configuration (see Section 6.3.1) as it denotes the

total amount of work-items per Work-Group. This means that for wavefront instances with fewer intra-wavefront points, we will inevitably have idle work-items. These work-items can be identified by a negative coordinate since all valid intra-tile coordinates are non-negative by default.

The intra-tile wavefront loop nest can be generated for the intra-tile space by hyperplane (6.5) using a polyhedral code generation tool [Bas04, QRW00, Che12]. The generated loop nest will include an outer wavefront loop and an inner nest of $d_t - 1$ loops that will place intra-tile coordinates in wavefront bucket $\mathcal{B}_I[w]$ for each wavefront instance $w \in \mathcal{W}_I$. Currently, these loop-nests have been hard-coded into the run-time system for 2D and 3D tiles.

6.3.1 1D or 2D Configuration

After completing the bucket population process we can proceed with the definition of a Work-Group and ND-Range configuration based on the total number of entries in each bucket.

For the ND-Range configuration the simplest approach is to create a 1D configuration defined as follows :

$$NDR(\mathcal{B}_{Tsize}, 1, 1) \tag{6.7}$$

However, the extent of each ND-Range dimension is typically associated with a maximum allowable value (e.g. 65536 for CUDA platforms). Consequently, if $\mathcal{B}_{Tsize}$ gets too large we might have to define a 2D configuration in order to respect the maximum extent limits. For that we can use a simple factorial algorithm to derive a parameter $\mathbf{f}_{BT}$ and our final 2D ND-Range configuration becomes :

$$NDR(\mathbf{f}_{BT}, \frac{\mathcal{B}_{Tsize}}{\mathbf{f}_{BT}}, 1) \tag{6.8}$$

For the work-group configuration we can use the same method to get a 2D configuration as

follows :

$$WG(\mathbf{f}_{BI}, \frac{\mathcal{B}_{Isize}}{\mathbf{f}_{BI}}, 1) \tag{6.9}$$

If however, we have up-to two parallel intra-tile dimensions we can simply produce the following work-group configurations :

$$WG(T_1, T_2, 1) \quad \text{or} \quad WG(T_1, 1, 1) \tag{6.10}$$

Throughout our experimental evaluation presented in Section 6.5 we only considered 1D work-group and nd-range configurations except for the programs that had a 2D rectangularly parallel intra-tile space amenable to a direct mapping based on (6.10).

6.4 Local Memory Management

According to the OpenCL execution model (Figure 2.6), each Work-Group is associated with a software managed memory space – called *Local Memory* – that physically resides in each compute unit (Figure 2.7). Because each work-group essentially represents a tile in the tile bucket $\mathcal{B}_T$, we seek a way of defining and managing local memory buffers in order to exploit intra-tile locality.

Since the total amount of work-groups is typically larger than the amount of compute units, local memory can be shared among multiple active work-groups. If the collective demand for local memory exceeds its physical capacity, the number of active work-groups per compute unit is reduced. This effect highlights the tight balance between locality and parallelism exemplified in Figure 6.8. In particular, we see that if the number of local memory buffers per work-group is reduced, then the number of active work-groups can be increased. On the other hand, if the local memory usage of a single work-group exceeds the physical capacity then the kernel invocation will fail completely. In an auto-tuning environment where tile-sizes can take arbitrary values

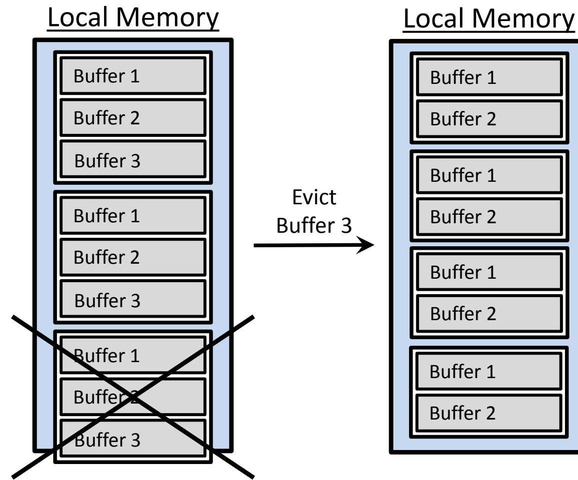


Figure 6.8: This figure highlights the trade-off between parallelism and locality. In particular, using one less buffer per work-group can activate one more work-group per compute-unit.

with a direct impact on buffer sizes, this situation can unnecessarily restrict the auto-tuning space. We overcome this problem by introducing an other run-time mechanism based on the concept of *Buffer Buckets* :

Definition 8. *A* Buffer-Bucket *denoted by $\mathcal{B}_B$, is a collection of abstract local-memory buffers accompanied by a kernel descriptor mapping the respective buffer-bucket to a specific device function.*

Each buffer-bucket is characterized by a tunable capacity parameter called *Local Memory Window* $\mathcal{L}_w$, that effectively represents the per-work-group availability of local memory. By introducing $\mathcal{L}_w$ we can dynamically control the number of local memory buffers per-work-group. In particular, local memory buffers can be added to a buffer-bucket at run-time as long as its contents do not exceed $\mathcal{L}_w$. If the $\mathcal{L}_w$ limit is reached the respective buffer-bucket is closed and no more additions can be performed. An obvious side-effect of this is that the order in which we add buffers to a buffer-bucket matters. For example, adding a series of small buffers and then a bigger one might have a different effect than adding the bigger buffer first and then the smaller ones. Furthermore, each addition is accompanied by a kernel descriptor mapping the contents of the respective buffer-bucket to a specific kernel function. The complete process is outlined by Figure 6.9. Notice that the kernel invocation in line 8 requires a buffer-bucket argument that specifies the kernel function to call and the total amount of local memory to be

```
1: initBufferBucket(𝓑_B,𝓛_w,Kernel(0))
2:
3: addBuffer(𝓑_B,B_1,Kernel(1))
4: addBuffer(𝓑_B,B_2,Kernel(2))
5:        ⋮
6: addBuffer(𝓑_B,B_n,Kernel(n))
7:        ⋮
8: invokeKernel(𝓑_B,𝓑_T,𝓑_I)
```

Figure 6.9: Mechanism for adding n abstract buffers $B_i : i \in [1..n]$ to a buffer-bucket $\mathcal{B}_B$ which is subsequently used for the kernel invocation. Note that the device code specified by the Kernel(0) descriptor will not use any of the buffers.

allocated dynamically. In other words a buffer-bucket constructs an execution environment in which the contained buffers are available for use.

The dynamic local memory management policy that we are proposing ranks the set of candidate local memory buffers and then utilizes the buffer-bucket abstraction and population mechanism (Figure 6.9) to construct an execution environment. This implies that $n + 1$ kernel versions are needed, where n is the total amount of buffers – $kernel_n$ will use all n buffers, $kernel_{n-1}$ will use the best $n - 1$ buffers according to their rank etc. In other words, buffers are added incrementally according to their rank and if the addition of buffer $B_i : i \in [1..n]$ results in exceeding $\mathcal{L}_w$ then all subsequent additions will fail and the kernel using $i-1$ buffers – indicated by the Kernel($i - 1$) descriptor – will be invoked. The ranking of the candidate local memory buffers is based on the following criteria [ALSU07] :

Temporal Reuse An array access exhibits sufficient temporal locality iff the rank of its access function is less than the dimensionality of the statement carrying the access.

Amount of Group-Reuse The total number of textual references to the same array with the same access function without considering any constant terms.

Self-Spatial Reuse This is more commonly known in recent HPC literature as *global memory access coalescing* and indicates whether an access function accesses a contiguous section of memory or not.

Each buffer entry contains the total size of the respective buffer and a set of parameters that are transferred to read-only constant memory and then used by pre-defined data-movement procedures to move data in and out of the buffers. More details about that will be discussed in the following sections.

We now proceed to explain how each abstract buffer entry is defined.

6.4.1 Buffer Definition

Let $\mathcal{F}_i$ be the multi-dimensional access function of array i, ignoring any constant terms. Furthermore, let C_i^t be a set of integers denoting the absolute distance between the maximum and the minimum constant terms across all textual references to array i for each dimension if i. We define buffer B_i of i to be the rectangular bounding box of $\mathcal{F}_i$ enlarged by the elements of C_i^t along each dimension, and characterized by the following parametric expressions :

Footprint Origins A set of parametric expressions denoted by $O_i(\vec{t}, \vec{T})$ for an array i, – where $\vec{t}$ and $\vec{T}$ the vectors of tile coordinates and tile sizes respectively – that represent the lexicographically minimum value of the access function $\mathcal{F}_i$ for each dimension under the domain of the tile $D_T(\vec{t})$ defined as :

$$D_T(\vec{t}) : \underbrace{t_i \cdot T_i}_{\mathbf{t}'_i} \leq x_i \leq \underbrace{t_i \cdot T_i}_{\mathbf{t}'_i} + T_i - 1 \qquad \text{for } i \in [1..d_t] \tag{6.11}$$

Evidently, the origin expressions depend on the tile coordinates and specify the position of the buffer elements in global memory.

Footprint Extents A set of parametric expressions – depending only on tile sizes – denoted by $E_i(\vec{T})$ for an array i, that represent the extent of the buffer's bounding box along each dimension. It is defined as the difference between the lexicographically minimum and maximum value of the access function $\mathcal{F}_i$ along each dimension under $D_T(\vec{t})$, incremented by the entries of C_i^t.

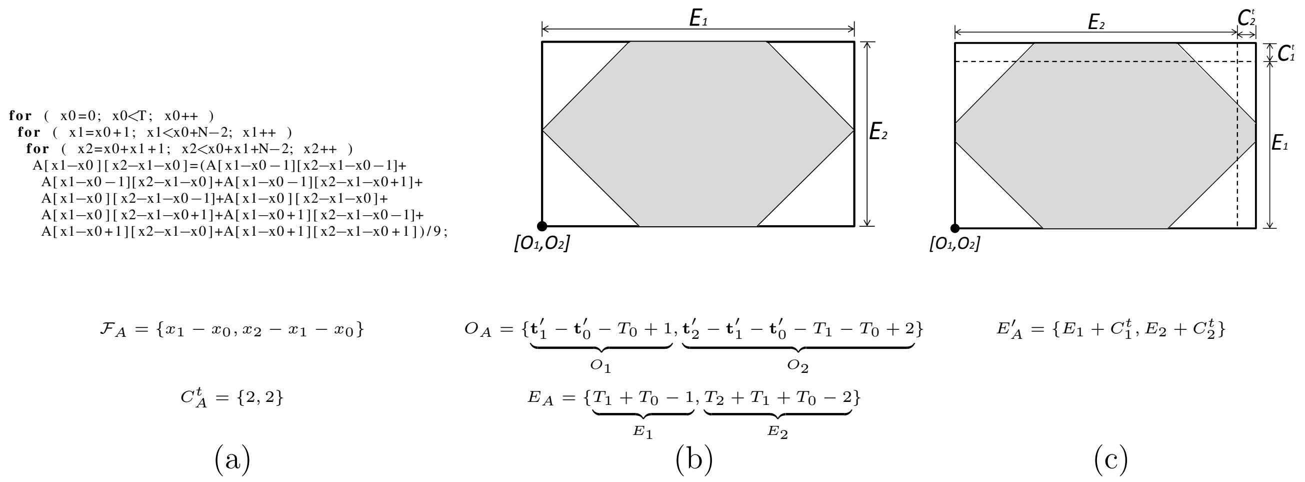


Figure 6.10: (a) A skewed Seidel-2d kernel (b) Buffer view of the access footprint of A if we ignore all constant terms (c) Buffer view of the access footprint of A if we consider all constant terms.

Notice that we only consider a single access function for each array implying that we ignore any arrays that have multiple linearly-independent access functions for any dimension. Figure 6.10 illustrates how buffer B_A is defined for the *Seidel-2D* kernel of 6.10(a), based on O_A, E_A and C_A^t.

Finally, given the total number of available threads $\mathcal{B}_{Isize}$, the buffer extents E_i, and a set of padding factors P_i, Algorithm 6 can be used to populate $\mathcal{B}_B$ with a 2D buffer entry identified by a set of 7 parameters; the buffer extents E_i, the padded buffer extents E_i', a buffer-specific thread layout captured by a width and a height parameter th_w^i and th_h^i respectively and the total amount of elements in the buffer E_{total}. When the buffer-bucket population process is over, all these parameters are transferred to read-only constant memory where they can be accessed by the parametric data-movement procedures discussed in Section 6.4.2. It is in this algorithm (line 12) that we can increase the number of threads in order to accelerate the data-movement process – an optimization similar to the one proposed by Bauer et al. [BCK11]. More details about that will be discussed in the following section.

6.4.2 Moving Data into and out of the Buffers

In the previous section we saw how the abstract buffers are defined and submitted into buffer-buckets for subsequent utilization. Now we are going to discuss how these buffers are utilized

Algorithm 6 Algorithm for adding a buffer entry to a buffer-bucket $\mathcal{B}_B$.

1: **procedure** AddBuffer2D($\mathcal{B}_B$,$\mathcal{B}_{Isize}$,E_i,P_i)
2: **if** ($\mathcal{B}_B$ is closed) **then** ▷ $\mathcal{L}_w$ has been exceeded by previous addition
3: **return**
4: **end if**
5: $E'_i[1] \leftarrow pad(E_i[1], P_i[1])$ ▷ Round height to be a multiple of $P_i[1]$
6: $E'_i[2] \leftarrow pad(E_i[2], P_i[2])$ ▷ Round width to be a multiple of $P_i[2]$
7: $E_{total} \leftarrow E'_i[1] \cdot E'_i[2]$ ▷ Total elements in buffer
8: $\mathcal{B}_{Bbytes} \leftarrow \mathcal{B}_{Bbytes} + E_{total}$ ▷ Total bytes to be dynamically allocated
9: **if** ($\mathcal{B}_{Bbytes} > \mathcal{L}_w$) **then**
10: Close $\mathcal{B}_B$ and **return** ▷ No more additions allowed
11: **end if**
12: Adjust $\mathcal{B}_{Isize}$ according to E_{total} or $E'_i[2]$ ▷ Optional
13: **if** ($E'_i[2] > \mathcal{B}_{Isize}$) **then**
14: $th^i_w \leftarrow \mathcal{B}_{Isize}$
15: **else**
16: $th^i_w \leftarrow E'_i[2]$
17: **end if**
18: $th^i_h \leftarrow \mathcal{B}_{Isize}/th^i_w$
19: $\mathcal{B}_B \leftarrow \{E_i, E'_i, th^i_h, th^i_w, E_{total}\}$ ▷ Add the buffer entry
20: $\mathcal{B}_{Bsize} \leftarrow \mathcal{B}_{Bsize} + 1$ ▷ Increment the buffer count
21: **return**
22: **end procedure**

by the run-time system.

Since the buffer extents as well as the work-group configuration are parametric, the movement of data in and out of the buffers needs to be parametric as well, i.e., data movement is carried out without any knowledge about the work-group and the buffer extents. For example, lets assume that tile sizes are known at compile-time. This means that the buffer and work-group extents are compile-time literals. If this is true then the compiler can take advantage of this knowledge and optimize the data movement code at compile-time. If for example a buffer has exactly the same extents as the respective work-group then the compiler can simply generate a single data-movement statement that is executed by each thread, i.e. all data items in the buffer are transfered in parallel. However, in our context this is not possible simply because tile sizes are parametric and therefore unknown at compile-time. Consequently, we propose a set of generic data movement methods in the form of run-time API functions that operate without knowing the buffer and work-group layouts. In particular, for a 2D array we propose Algorithms 7 and 8 for moving data in and out of the buffers respectively. In both algorithms we use two loops (lines 4 and 6) because we assume that the work-group extents are always smaller than the buffer extents which is the most general case.

Algorithm 7 The basic move-in procedure for a 2D buffer B_i executed on the device code by each thread.

```
1:  procedure MoveIn2DGeneric(O_i, C_i^n, B_i, b_i)
2:      l_1 ← wi.x/th_w^i,   l_2 ← wi.x mod th_w^i              ▷ Recover new thread layout
3:      g_1 ← O_i[1] − C_i^neg[1] + l_1,   g_2 ← O_i[2] − C_i^neg[2] + l_2   ▷ Recover global position of buffer
4:      while (g_1 < b_i[1]) and (l_1 < E'_i[1]) do              ▷ Height traversal
5:          if (g_1 ≥ 0) then
6:              while (g_2 < b_i[2]) and (l_2 < E'_i[2]) do      ▷ Width traversal
7:                  if (g_2 ≥ 0) then
8:                      buffer[l_1][l_2] = global[g_1][g_2]
9:                  end if
10:                 g_2 ← g_2 + th_w^i
11:                 l_2 ← l_2 + th_w^i
12:             end while
13:         end if
14:         g_1 ← g_1 + th_h^i
15:         l_1 ← l_1 + th_h^i
16:     end while
17: end procedure
```

Algorithm 8 The basic move-out procedure for a 2D buffer $\mathbb{B}_i$ executed on the device code by each thread.

```
1:  procedure MoveOut2DGeneric(O_i, C_i^n, B_i, b_i, F_i(t_1, t_2))
2:      l_1 ← wi.x/th_w^i,   l_2 ← wi.x mod th_w^i              ▷ Recover new thread layout
3:      g_1 ← O_i[1] − C_i^neg[1] + l_1,   g_2 ← O_i[2] − C_i^neg[2] + l_2   ▷ Recover global position of buffer
4:      while (g_1 < b_i[1]) and (l_1 < E_i[1]) do       ▷ Height traversal using original buffer height
5:          if (g_1 ≥ 0) then
6:              while (g_2 < b_i[2]) and (l_2 < E_i[2]) do   ▷ Width traversal using original buffer width
7:                  if (g_2 ≥ 0) and (F_i(l_1, l_2)) then
8:                      global[g_1][g_2] = buffer[l_1][l_2]
9:                  end if
10:                 g_2 ← g_2 + th_w^i
11:                 l_2 ← l_2 + th_w^i
12:             end while
13:         end if
14:         g_1 ← g_1 + th_h^i
15:         l_1 ← l_1 + th_h^i
16:     end while
17: end procedure
```

Evidently this approach is not ideal because the two loops in lines 4 and 6 yield a considerable overhead. In order to avoid this overhead we can dynamically (at run-time) enforce a specific work-group layout that will allow us to eliminate one or both of these loops. This can be done in line 12 of Algorithm 6 by increasing the number of work-items in the work-groups, if necessary, in order to enforce a specific relation between the work-group and the buffer. This will enable us to use faster data movement methods for the price of potentially more work-items that are not used for computation. In fact, for most SCoPs in practice we can guarantee that the condition of line 13 in Algorithm 6 is true without increasing the number of work-items which allows us to eliminate the loop of line 6 at no cost. An additional optimization for Algorithms 7 and 8 takes advantage of the full-tile separation mechanism of Algorithm 5. In particular the bounds conditions in lines 4, 5, 6 and 7 can be avoided if we are dealing with a full tile.

If line 12 of Algorithm 6 does lead to an increase in work-items then we need to make sure that the additional work-items do not interfere with our computation (because we are only using them for data movement). For that purpose we map the extra threads to -1 entries in the thread bucket (just like we do for intra-tile wavefronts) and protect our computation with special if-guards that filter away threads with negative thread-bucket entries. Keep in mind that for intra-tile wavefronts this filtering is already in place and embodied by the wavefront conditions (Figure 6.5(c)).

The main difference between the move-in and move-out procedures lies on line 7. In particular, the condition of line 7 is amended with $F_i(\vec{l_i})$, a conditional expression that depends on the buffer coordinates $\vec{l_i}$ and restricts the move-out procedures to operate only on the elements that have actually been written by the respective tile (the grey area in Figure 6.10). Therefore, in order to determine $F_i(\vec{l_i})$ we need to examine the write accesses of the program.

Let N_i be the total number of textual references to array i that perform a write operation and $C^j_{i_{write}}$ the constant term for each reference $j \in N_i$. Additionally, if we solve $O_i = 0$ for $\vec{t}$ and replace $\vec{t}$ in (6.11) with the solution, we end up with the domain of the buffer :

$$D_{T_i} \cdot \begin{pmatrix} \vec{l_i} \\ \vec{T} \\ 1 \end{pmatrix} \geq 0 \tag{6.12}$$

where $\vec{l_i}$ is the vector of buffer coordinates and $\vec{T}$ the vector of tile sizes. Each write operation $j \in N_i$ can now be expressed in terms of $\vec{l_i}$ by using a polyhedral code generation tool (e.g. CLooG) to scan D_{T_i} under the affine transformation $\mathcal{F}^j_{i_{write}}$ defined as :

$$\mathcal{F}^j_{i_{write}} = \mathcal{F}_i + (C^j_{i_{write}} - C^n_i) \tag{6.13}$$

The final solution is the convex hull D_{CH_i} of all $\mathcal{F}^j_{i_{write}} : j \in [1..N_i]$ under D_{T_i} which is the collective footprint of all write operations to buffer B_i, expressed in terms of $\vec{l_i}$. In order to retrieve the solution and construct the conditional expression $F_i(\vec{l_i})$, we first need to convert the syntax trees produced by the polyhedral code generation tool back into matrix forms and then perform the convex hull operation. Finally, we convert the resultant convex hull matrix D_{CH_i} into the symbolic expression $F_i(\vec{l_i})$ and use it as an argument to the respective move-out procedures. The complete process is summarized by the following Algorithm:

Algorithm

Input : The set of polyhedral access functions $\mathcal{F}$ of the program. The symbolic name of an array i.

Output : A conditional expression $F_i(\vec{l_i})$ that would restrict move-out procedures to operate only on the elements of array i that where actually written by the respective tile.

Step 1 Using the symbolic name of i retrieve $\mathcal{F}_i$ and C^n_i from $\mathcal{F}$.

Step 2 Determine $C^j_{i_{write}}$ for each write reference $j \in [1..N_i]$ of i, where N_i the total number of textual references to i that correspond to a write operation.

Step 3 Calculate the origin expressions O_i of i using $\mathcal{F}_i$ and (6.11).

Step 4 Solve the system $O_i = 0$ for $\vec{t}$ and substitute $\vec{t}$ in (6.11) with the solution to get the buffer domain D_{T_i}.

Step 5 Construct the affine transformations $\mathcal{F}^j_{i_{write}}$ for each write access $j \in [1..N_i]$ using basic matrix manipulation.

Step 6 Use a polyhedral code generation tool (e.g. CLooG) to scan D_{T_i} under each write transformation $\mathcal{F}^j_{i_{write}} : j \in [1..N_i]$.

Step 7 Process the syntax trees produced by **Step 6** and create the footprint domains $D^j_{F_i}$ for each write access $j \in [1..N_i]$ of array i.

Step 8 Use a polyhedral library to get the convex hull of all $D^j_{F_i} : j \in [1..N_i]$ denoted by D_{CH_i}.

Step 9 Construct one conditional expression for each row in D_{CH_i} and omit the ones that don't involve more than one coordinate variables.

Step 10 Construct a conjunction of all conditions from **Step 9** and return the resultant conditional expression.

Implementation

First of all, the access functions of the program $\mathcal{F}$ can be retrieved in the form of `AccessPattern` objects for each `Datum` in a `RosePollyModel` symbol table. Because each `Datum` has a symbolic name, we can simply get the `AccessPattern` objects that belong to a `Datum` with name i. We can then group all `AccessPattern` objects based on whether they are write or read patterns and concentrate on the write ones. Afterwards, we can get the `pollyMap` objects of the write patterns and request a `simple_matrix` that represents the actual access function. We know that in these simple matrices the last column is the constant term of the access function so from that column we can extract C^n_i and $C^j_{i_{write}}$ (Steps 1 and 2).

For Step 3, extracting the origin expressions for i is simple. We first get a `simple_matrix` from any `AccessPattern` of i and ignore any values on the last column. We then replace any

positive entries with $\mathbf{t}'_i$ (i.e. $t_i \cdot T_i$) and negative entries with $-\mathbf{t}'_i - T_i + 1$ (i.e. $-t_i \cdot T_i - T_i + 1$). Obviously in order to do such substitutions we need to adjust our simple matrix by adding columns for the tile sizes and tile coordinates. The origin expressions for i are now captured by a `simple_matrix` object.

In Step 4 we can construct a `pollyMap` from the `simple_matrix` of step 3 and invoke a linear equation solver in order to get the solution of $O_i = 0$. We now construct a new `simple_matrix` that will represent the domain of the buffer D_{T_i} by substituting the solution of $O_i = 0$ to (6.11) again by manipulating `simple_matrix` objects.

For steps 5, 6 and 7 we can easily construct `simple_matrix` objects for each $\mathcal{F}^j_{i_{write}} : j \in [1..N_i]$. and D_{T_i} (we already have D_{T_i} from step 4) and then instantiate `pollyMap` and `pollyDomain` objects respectively that we can supply to `RoseCloog` in order to get `pollyDomain` objects back representing $D^j_{F_i}$. Notice that RosePolly allows us to hide the details of this transition behind the `RoseCloog` class. As a result, we do not need to worry about invoking CLooG, and manipulating syntax trees or any cloog-specific data structure that would contaminate our implementation with third-party code. Getting the convex hull for step 8 is now a simple incremental invocation of the `pollyDomain::convex_hull` method.

Finally for steps 9 and 10 we can create our symbolic conditional expressions by examining the `simple_matrix` that represents the convex hull of step 8. In particular, for each row we can construct a syntax tree of an inequality with all non-zero entries in that row being placed on the left hand side and zero on the right hand side. We can then create the final disjunction of all the individual row conditions.

6.5 Putting It All Together

By putting all the pieces together we can formulate a complete GPU code-generation algorithm for static control programs. This algorithm would produce two pieces of code: (i) the GPU mapping code that runs on the host side, and (ii) the $n+1$ kernels (where n is the total number of local memory buffers) that are executed on the device. Figures 6.11 and 6.12 show the

```
1: Host-Code
2: InitBufferBucket(B_B,L_w,Kernel(0))
3: InitThreadBucket(B_I)
4: if (Intra-tile wavefront) then
5:     SetIntraWave(B_I,d_t)
6: else
7:     SetRectangularLayout(B_I,T_1,...,T_{d_par})
8: end if
9: AddBuffer(B_B,B_Isize,E_1,P_1,Kernel(1))
10:    ⋮
11: AddBuffer(B_B,B_Isize,E_n,P_n,Kernel(n))
12: if (Tile wavefront) then
13:     for each w ∈ W do
14:         for each loop L_wi in S_WT(w) do
15:             PopulateTileBucket(B_T,t_i ∈ L_wi)
16:         end for
17:         InvokeKernel(B_T,B_I,B_B)
18:     end for
19: else
20:     for each loop L_i in S_TP do
21:         PopulateTileBucket(B_T,t_i ∈ L_i)
22:     end for
23:     InvokeKernel(B_T,B_I,B_B)
24: end if
```

Figure 6.11: Template of the produced host-code.

templates of the produced host and device code both of which provide a clear guide for the code generation algorithms.

All capitalized functions in Figures 6.11 and 6.12 constitute the platform-independent runtime environment[6] that supports the GPU mapping mechanisms as well as the data-movement procedures and the tile/intra-tile recovery methods that reside on the device code. In particular, the latter are using the built-in **glw** and **gli** index variables to access the tile and thread-bucket entries which have been transferred to concurrent data structures by the host code. More specifically, the tile-bucket entries are stored in global memory and the thread-bucket entries are stored in image-memory while the buffer-bucket entries are stored in constant memory. The condition in Line 29 of the intra-tile wavefront code simply checks whether the corresponding thread-bucket entry is negative or not. The same condition is found on non-wavefront device code as well in case we have increased the number of threads to facilitate more efficient data-movement procedures.

[6]Currently supporting CUDA targets.

```
 1: Rectangularly Parallel Intra-Tile Execution
 2: RecoverTileCoordinates
 3: RecoverIntraTileCoordinates   ▷ The parallel ones
 4: for each sequential tile loop do
 5:     Synchronize
 6:     MoveIn(1,O_1,C_1^neg,b_1)
 7:       ⋮
 8:     MoveIn(n,O_n,C_n^neg,b_n)
 9:     Synchronize
10:     if (ValidThread) then   ▷ Optional
11:         if (FullTile) then
12:             P_intra^full(n)       ▷ Computation with n buffers
13:         else
14:             P_intra(n)            ▷ Computation with n buffers
15:         end if
16:     end if
17:     Synchronize
18:     MoveOut(1,O_1,C_1^neg,b_1,F_1)
19:       ⋮
20:     MoveOut(n,O_n,C_n^neg,b_n,F_n)
21: end for
22:
23: Intra-Tile Wavefront Execution
24: RecoverTileCoordinates
25: MoveIn(1,O_1,C_1^neg,b_1)
26:   ⋮
27: MoveIn(n,O_n,C_n^neg,b_n)
28: for each intra-tile wavefront instance w do
29:     if (ValidThread) then
30:         RecoverIntraTileCoordinates
31:         if (FullTile) then
32:             P_intra^full(n)       ▷ Computation with n buffers
33:         else
34:             P_intra(n)            ▷ Computation with n buffers
35:         end if
36:     end if
37:     Synchronize
38: end for
39: MoveOut(1,O_1,C_1^neg,b_1,F_1)
40:   ⋮
41: MoveOut(n,O_n,C_n^neg,b_n,F_n)
```

Figure 6.12: Template of the produced device code.

The complete code generation algorithm has not been implemented yet. For our experimental evaluation we produced GPU code manually based on the code templates of Figures 6.11 and 6.12. As a result, various components had to be acquired independantly and then assembled together into the final form of the code. More specifically, for Algorithm 6.2.1 we used an existing implementation provided by PoCC (called `ptile`) while Algorithms 4 and 6.4.2 were implemented using RosePolly as we described in the respective implementation sections.

6.6 Experimental Evaluation

The purpose of our experimental evaluation is to assess the following two main properties of our code generation method:

- In the presence of rectangularly-parallel tile and intra-tile spaces that do not require wavefront parallelism, we would like to evaluate the overhead induced by our run-time system. Such study will also tell us to what extent we can use run-time tuning as a substitute of iterative compilation.
- In the presence of wavefront parallelism, we would like to assess the effectiveness of our run-time system in mapping wavefronts of tile and intra-tile points on a GPU execution environment.

For both experiments we compared our solution to a state-of-the-art compile-time method with no run-time support called *PPCG* [VCJC+13](version c7179a0). PPCG utilizes polyhedral analysis and code-generation for producing statically tiled CUDA code, i.e., tile-sizes are known at compile-time. In both systems, the Pluto [BBK+08b, BR07] scheduling algorithm is used to enable tiling through affine transformations. Furthermore, in order to isolate the performance effect of our run-time system we disabled privatization on all experiments; an optimization – provided by default on PPCG – that utilizes registers to perform loop unrolling, an operation equivalent to thread-merging [YXKZ10]. This will not affect our evaluation since privatization is an independent optimization that can be investigated in a separate study.

	Compute Units	Processing Elements (Cores)	Local Memory (KB)	Peak Bandwidth (GB/s)	Peak Compute Performance (GFLOPS-SP)	Compute Capability	CUDA
GTX 280	10	240	16	141.7	622.1	1.3	4.2
GT 540M	2	96	48	28.8	258	2.1	4.2
GTX 580	16	512	48	192	1581	2.1	4.2
M2070	14	448	48	150	1030	2.0	4.2.9
K20c	13	2496	48	208	3524	3.5	5.0.35

Table 6.1: Compute and memory characteristics of the GPUs used in the experimental evaluation.

Our experiments were conducted on a variety of NVIDIA GPU devices spanning from the early GTX 280 with 1.3 compute capability to the latest high-end K20c with 3.5 compute capability. Table 6.1 shows a complete list of the GPUs used along with a set of key attributes.

With respect to our first assessment, we used the well-known matrix-multiplication example as a representative of rectangularly-parallel programs. On the other hand we used 5 stencil programs from the polybench suite[7] for our second assessment. In particular, we used the *ADI* and *Seidel-2d* benchmarks that utilize the thread-bucket mechanism for intra-tile wavefront execution and the *Jacobi-1d*, *Jacobi-2d* and *fdtd-2d* benchmarks that utilize Algorithm 4 to avoid intra-tile wavefronts.

In most of our graphs we used normalized metrics (i.e. execution time for first assessment and GFLOP performance for our second assessment) simply because the absolute values offer no value to our study. In fact, normalized values are more convenient because the reader can make relative percantage-based comparisons easily. The reasons why we do not use aboslute values are the following:

- For our first assessment we want to measure the run-time overhead of our system and the correlation between run-time and compile-time tuning. We do not evaluate (or propose) any novel optimization for this kind of programs and therefore we do not anticipate speed-ups. In fact, we could use any optimized implementation of matrix-multiply in our experiment (e.g. from CUBLAS[8]) but we chose PPCG because our primary objective is to evaluate the potential for performance portability through a complete source-to-source compilation path.

[7] www.cs.ucla.edu/~pouchet/software/polybench/

[8] developer.nvidia.com/cuBLAS

- As opposed to our first assessment, in our second one we do want to evaluate a new optimization, i.e. the tile-bucket and thread-bucket mechanisms for executing wavefront parallel programs on GPUs. In this case we do not care about absolute performance because PPCG is the state-of-the-art source-to-source method for automatically generating GPU code for wavefront parallel SCoPs. To the best of our knowledge there is no better alternative to date.

The results are grouped based on GPU model and they can be distinguished into two kinds of graphs per GPU. The first kind is related to our first assessment and includes six line graphs per GPU: (a), (b), (c), (d), (e) and (f). In four of these graphs (i.e., graphs (a), (c), (e) and (f)) each point represents the normalized execution time for a single tile-size configuration (each point comes from an average of 10 runs) while two additional graphs (i.e., graphs (b) and (d)) are used to show the relative run-time overhead for each tile-size configuration along with the overall average overhead.

The second kind is related to our second assessment and consists of a single bar diagram per GPU, showing the best performance found within a given search space of tile-sizes. The additional bars per benchmark shown in those diagrams show the respective performances when using less local memory buffers (the far right bar denote the performance when no buffers are used) in an attempt to highlight the importance of exploring the locality/parallelism trade-off discussed in Section 6.4.

We now proceed to discuss the conclusions with respect to our two assessments.

Assessment 1 : Run-time overhead and correlation between compile-time and run-time tuning

For our first assessment graphs (a) and (c) demonstrate the correlation between our method (ptileGPU line) and ppcg accross a set of tile-size configurations without and with local memory utilization respectively. On the other hand, graphs (e) and (f) show the effect of using thread-buckets. By using thread-buckets to carry intra-tile coordinates we are able to evaluate two

main properties of our system: (i) the cost of using the thread-bucket mechanism, (ii) the effectiveness of the thread-bucket mechanism in enabling data-movement optimizations. More specifically, if we use thread-buckets then we are able to increase the number of threads beyond the ones needed for computation. The additional threads can then be used to optimize the data-movement procedures by eliminating both loops in Algorithms 7 and 8. The remaining graphs (b) and (d) show the relative run-time overhead for each configuration point of graphs (a) and (c) respectivelly as well as the average overhead (dashed red line).

For each experiment our tile-sizes ranged from 8 to 32 with a stride of 4 for each of the 3 dimensions of matrix-multiplication. This yielded a total of 343 configuration points for each experiment arranged lexicographically. Because the number of configurations was very large we removed some of them from the graphs – with no impact on our conclusions – in order to make them more easily readable. The configuration points we removed were the following:

- Size 8, 12, 20 and 28 for the innermost tile loop.
- The pairs of (20,16) (24,16) (28,16) (32, 16) tile sizes for the two inner tile loops (16 for the innermost).
- The pairs of (28,24) and (32,24) tile sizes for the two inner tile loops (24 for the innermost).

Of course, in the calculation of the average overhead – shown as a red dashed line in graphs (b) and (d) – we considered the entire range of tile-sizes including the ones we removed from the graphs for clarity.

Our first conclusion with respect to graphs (a) and (c) is that our run-time tuning method (ptileGPU line) correlates with the iterative compilation method – embodied by the ppcg line – to a satisfactory degree. In other words it can be considered reasonable to avoid iterative compilation and use our method for run-time performance tuning instead. In fact, Table 6.2 shows that the time saved for searching 343 configurations that use shared memory is considerable.

Perhaps the most striking exceptions are the following:

- Configurations 6, 21, 37, 51, 64, 75 and 84 on graph (c) of GTX 280
- Configurations 89 and 104 on graph (a) of K20c

These configurations could be the starting point of a more detailed study that would help us attain a better understanding of the differences between the two tuning methods.

With respect to the induced run-time overhead, graphs (b) and (d) show that the relative run-time overhead exhibits a normal flactuation around the average which is not always low. In fact, it reaches 30.1% on the GTX 280 and 10.3% on the K20c. Understanding the source of this overhead and attempting to minimize it could be the subject of future work. An interesting observation towards that direction is that the average overhead increases if we move from graph (b) to (d). This effect was actually expected since the parametric nature of the data-movement procedures makes them conservative and therefore less efficient (this effect is not so clear on the GT 540M for reasons that are not yet fully understood). Consequently, minimizing the average overhead for graph (d) could be achieved by improving the efficiency of the built-in data movement procedures. Finally, it is worth noting that in some cases (GTX 580 and K20c) the average overhead on graph (b) is negative which could be attributed to the performance improvement due to the run-time full-tile separation optimization.

If we now move to graphs (e) and (f) it is clear that the additional cost of using thread-buckets dominates the performance of our run-time method making it significantly slower than ppcg. However, on the fourth graph we see that much of that performance loss is recovered if we enable local memory. Strangely though, the unrolling optimization for the data-movement procedures appears to have a negligible impact on performance (i.e. the green and blue lines almost coincide) which is apparent across all GPUs of our experiment.

Assessment 2 : Effectiveness of run-time system in mapping tile and intra-tile wavefronts

For the second assessment we present one bar diagram for each device showing the best performances found within a given search space. These search spaces were not equivalent between the

two methods, i.e., compile-time and run-time simply because each method embodies a different strategy in mapping the respective wavefronts. For example, in all benchmarks used in our experiments PPCG (the compile-time method) did not tile the outer-most (time) loop in order to minimize tile wavefronts or completely avoid wavefronts for the Jacobi and fdtd benchmarks.

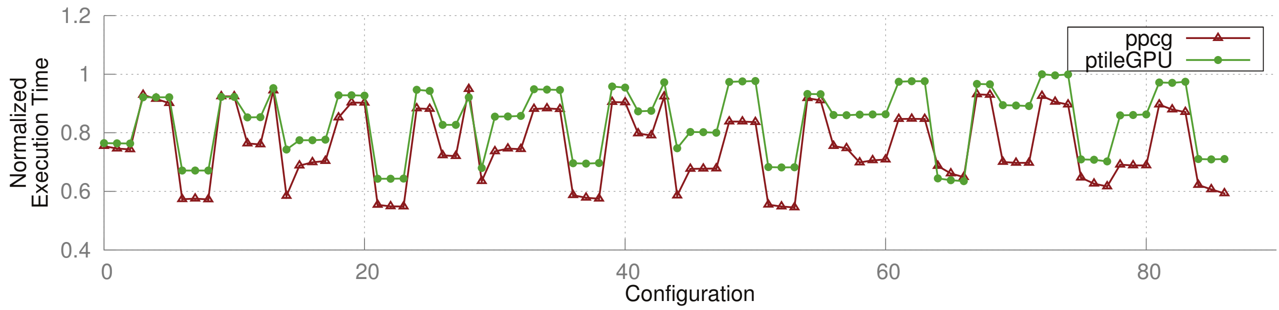
For all our benchmarks we used the same search space we used for matrix multiplication i.e. tile sizes ranging from 8 to 32 with a stride of 4. The only exception was the Jacobi-1d benchmark where we observed a clear linear increase in performance as the outer (time) tile size increased up to 225. Such exploration was not possible for PPCG simply because the outer time dimension was not tiled. This effect was not observed in the rest of the benchmarks primarily because of the effect of local memory in performance. In particular, on those benchmarks (i.e. ADI, Jacobi-2d, fdtd-2d and Seidel-2d) local memory buffers increase proportionally in size along each dimension as the time-tile size increases. Consequently, large time-tile sizes have a clear negative impact on occupancy as well as data-movement cost.

By looking at all the bar diagrams it is clear that our run-time system is effective in mapping tile and intra-tile wavefronts on GPUs. Furthermore, the additional bars per-benchmark show that in some cases using fewer local memory buffers yield better performance as we expected and already explained in Section 6.4.

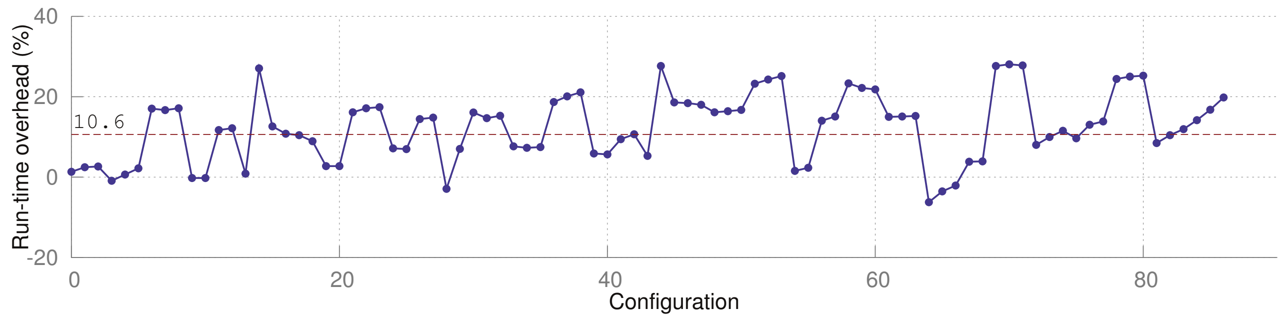
More specifically we see that for Jacobi-2d, using 2 buffers is never beneficial. In fact, for the GT540M and GTX580 it is better if we don't use local memory at all. This observation doesn't mean that local memory buffers do not yield a performance improvement in general, i.e. for all tile-size configurations. Instead it means that for some combination of tile-sizes using one or no local memory buffers results in the best performance within our search space. There are two main reasons for this. First of all the buffer definitions for Jacobi-2d include a coefficient of 2 (this coefficient is a result of the loop skewing transformation performed by Pluto in order to enable tiling) for the time-tile size along each dimension. This means that if we increase the time-tile size local memory usage puts a considerable pressure on occupancy and data-movement cost. Secondly, the data-movement associated with array B is largely redundant because only a few elements of B are actually communicated between wavefronts thus moving

the entire footprint of B in and out of local memory is wasteful to a large degree. Since array B is ranked second against array A the blue bar clearly shows that redundantly moving B in and out of local memory hurts performance. This could be avoided with a precise polyhedral analysis of B similar to [Grö09], an overlapped tiling method [HPS12, KBB+07] or a split-tiling method [GCK+13]. The first one has not proven yet to be efficient for 2D problems while the later two are restrictive to a small subset of SCoPs.

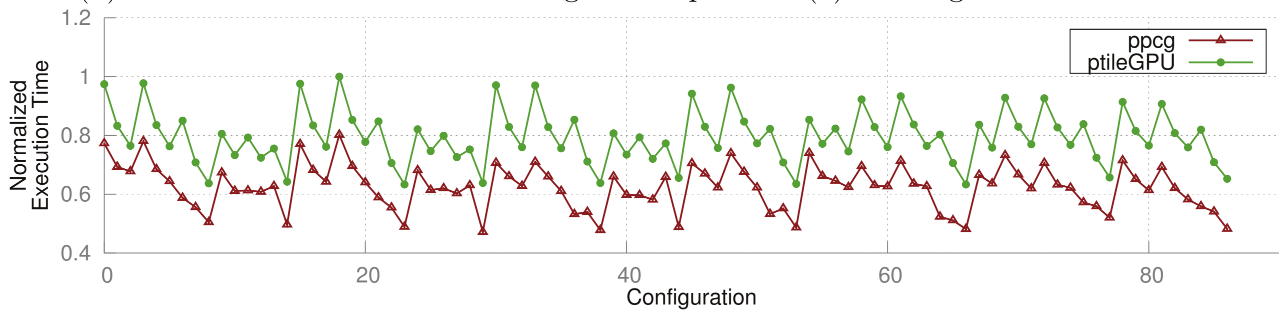
GTX280



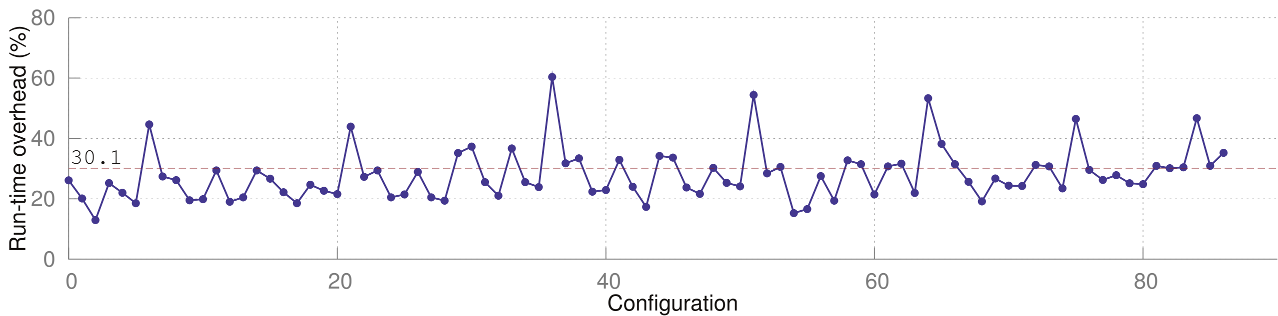
(a) No local memory used. Worst execution time 0.702*sec*.



(b) Relative overhead for each configuration point of (a). Average overhead 10.6%.

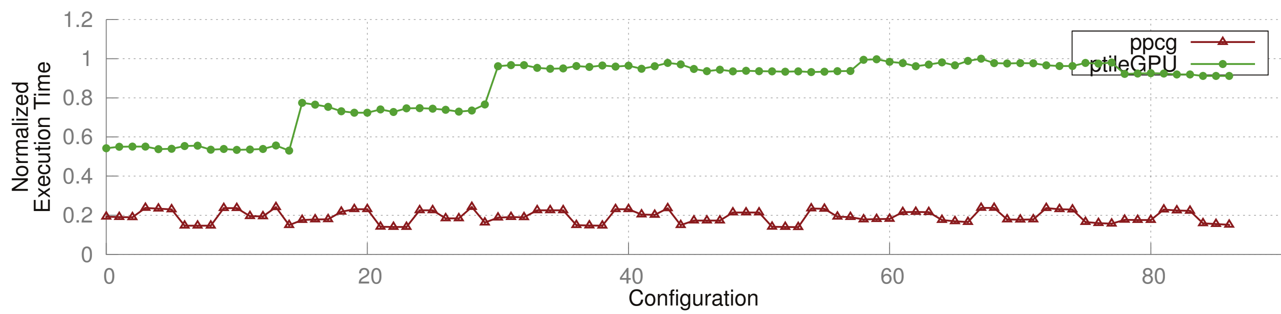


(c) With local memory. Worst execution time 0.612*sec*.

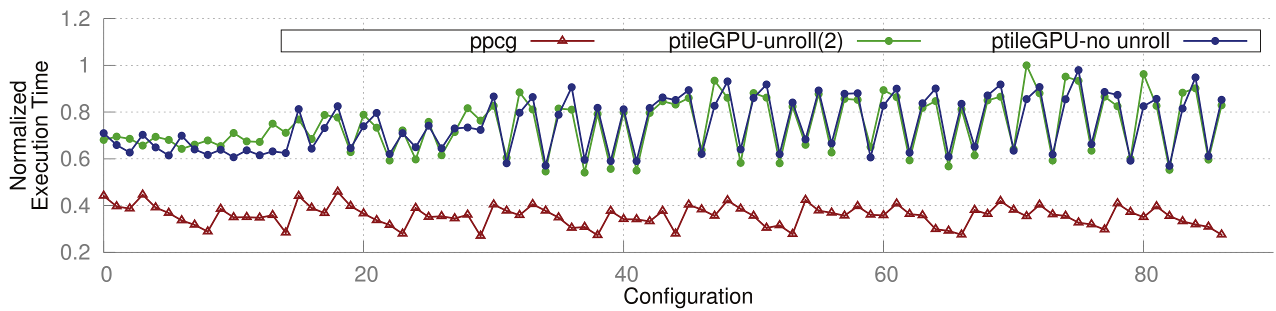


(d) Relative overhead for each configuration point of (c). Average overhead 30.1%.

Figure 6.13: Performance profiles and relative overhead of matrix-multiplication on GTX280 for a $2k \times 2k \times 2k$ problem without using thread buckets. Each point of (a) and (c) is normalized with worst execution time. The dashed red lines of (b) and (d) represent the global average run-time overhead for the entire configuration space.



(e) No local memory used. Worst execution time 2.573.



(f) With local memory and unrolling optimization. Worst execution time 0.759sec.

Figure 6.14: Performance profiles of matrix-multiplication on GTX280 for a $2k \times 2k \times 2k$ problem using thread buckets. Each point of (a) and (b) is normalized with worst execution time.

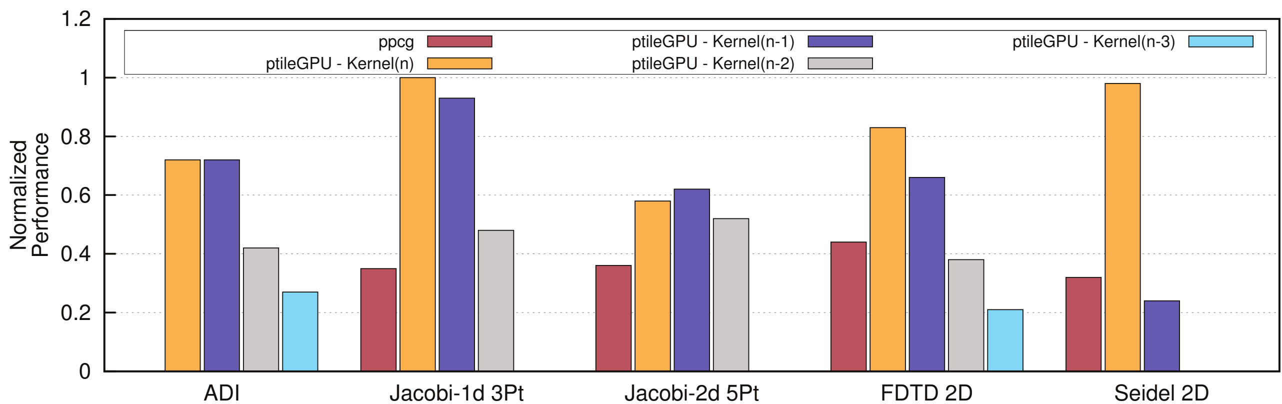
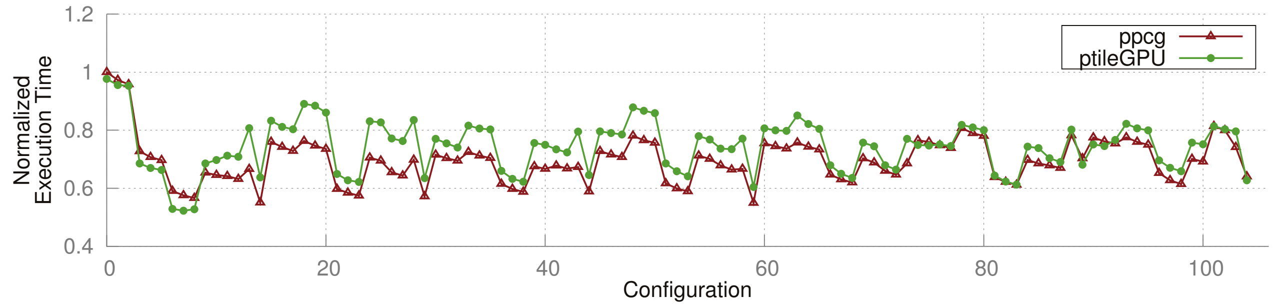
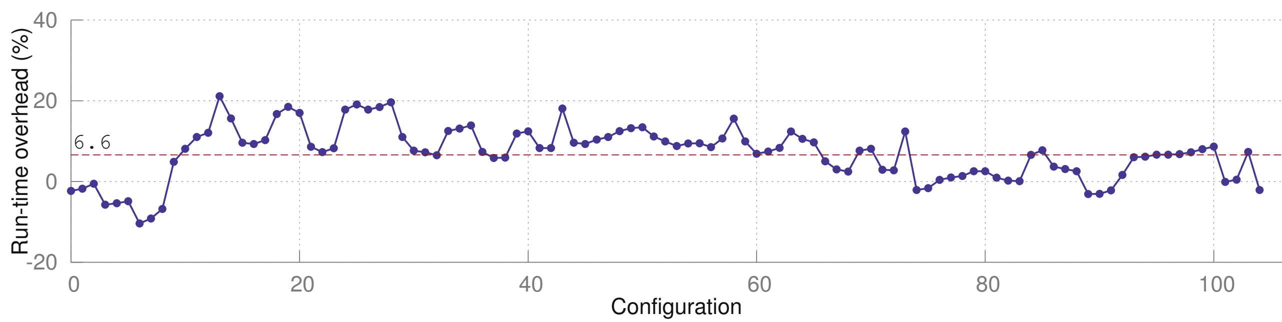


Figure 6.15: Best performances found after tile-size search on GTX280. Performances are normalized with the best performance found which was 17.08 GFLOPs for Jacobi-1d. The ADI benchmark does not include a red bar because the respective PPCG code yielded incorrect code (out-of-bounds access).

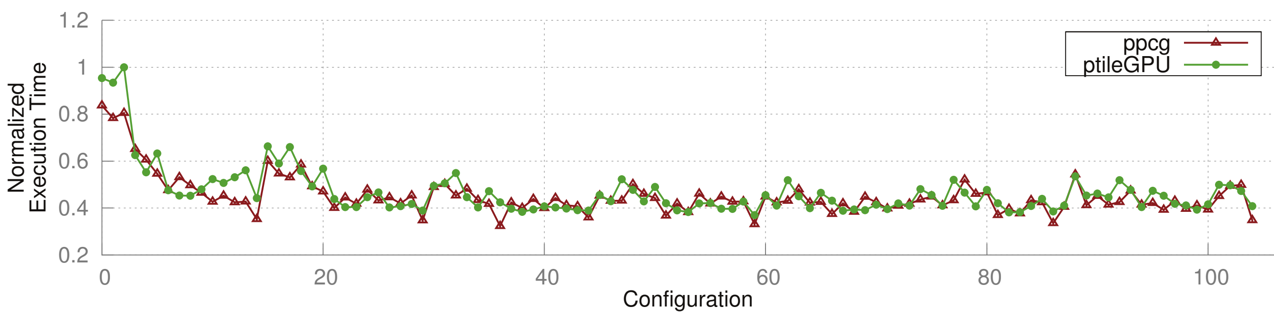
GT540M



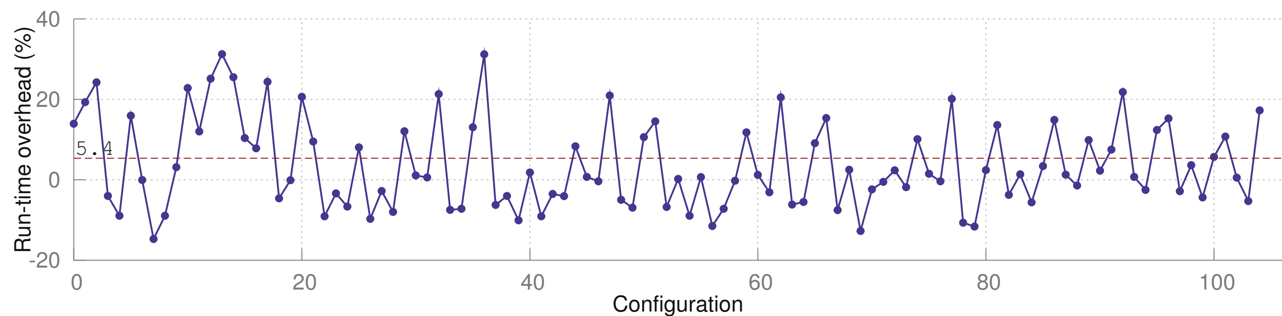
(a) No local memory used. Worst execution time 2.909*sec*.



(b) Relative overhead for each configuration point of (a). Average overhead 6.6%.

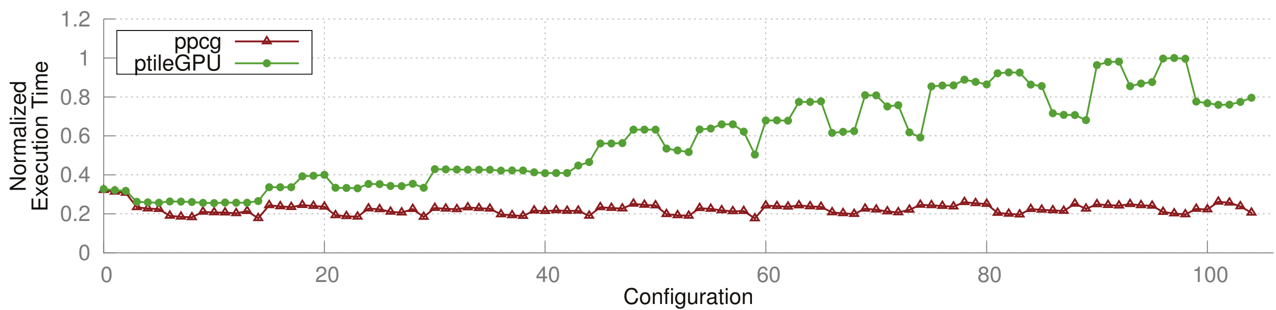


(c) With local memory. Worst execution time 2.467*sec*.

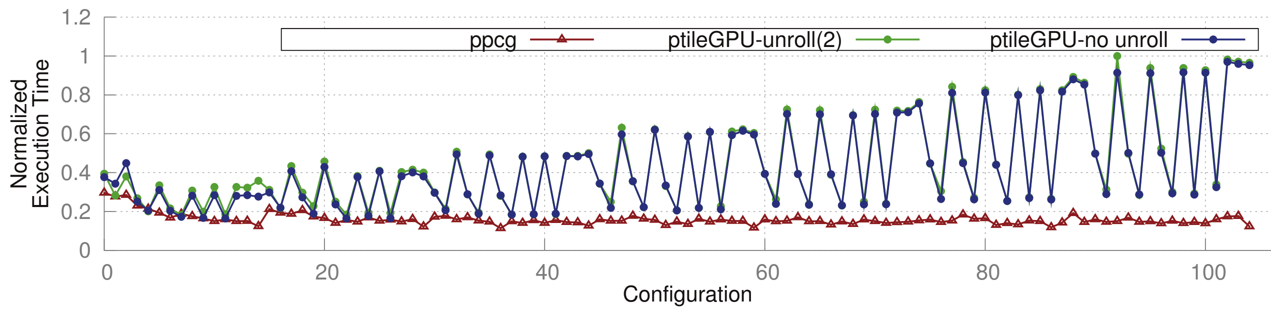


(d) Relative overhead for each configuration point of (c). Average overhead 5.4%.

Figure 6.16: Performance profiles and relative overhead of matrix-multiplication on GT540M for a $2k \times 2k \times 2k$ problem without using thread buckets. Each point of (a) and (c) is normalized with worst execution time. The dashed red lines of (b) and (d) represent the global average run-time overhead for the entire configuration space.



(e) No local memory used. Worst execution time 8.342*sec*.



(f) With Local memory and unrolling optimization. Worst execution time 6.969*sec*.

Figure 6.17: Performance profiles of matrix-multiplication on GT540M for a $2k \times 2k \times 2k$ problem using thread buckets. Each point of (a) and (b) is normalized with worst execution time.

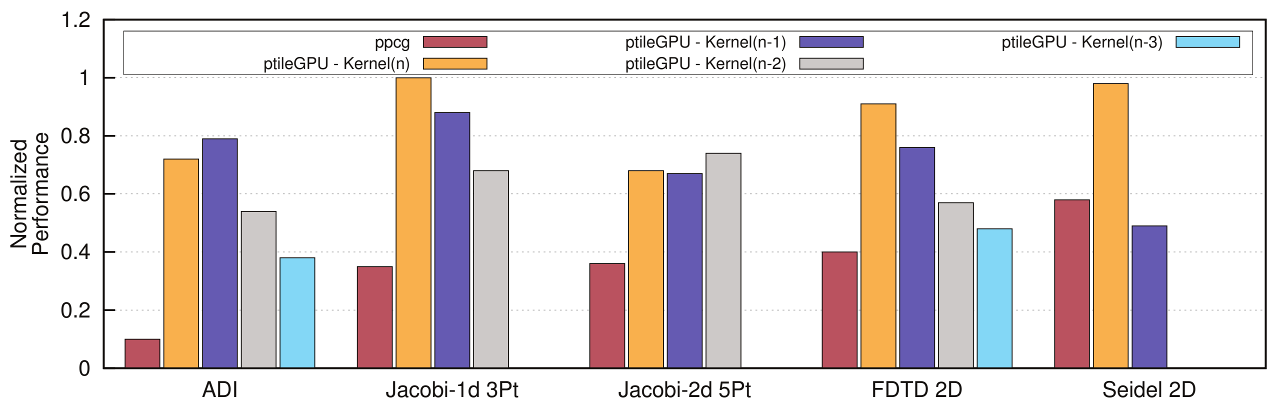
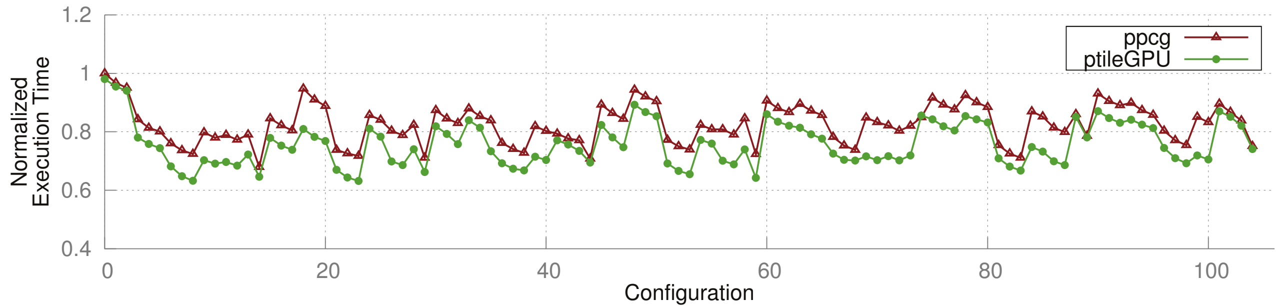
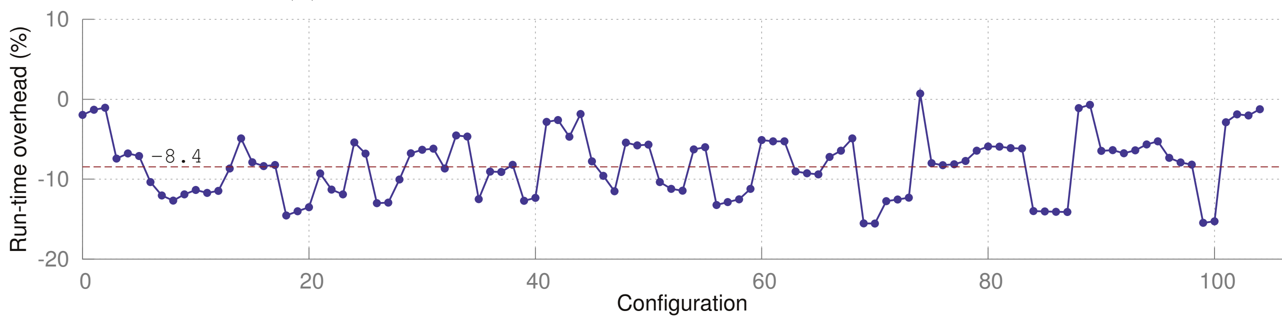


Figure 6.18: Best performances found after tile-size search on GT540M. Performances are normalized with the best performance found which was 4.13 GFLOPs for Jacobi-1d.

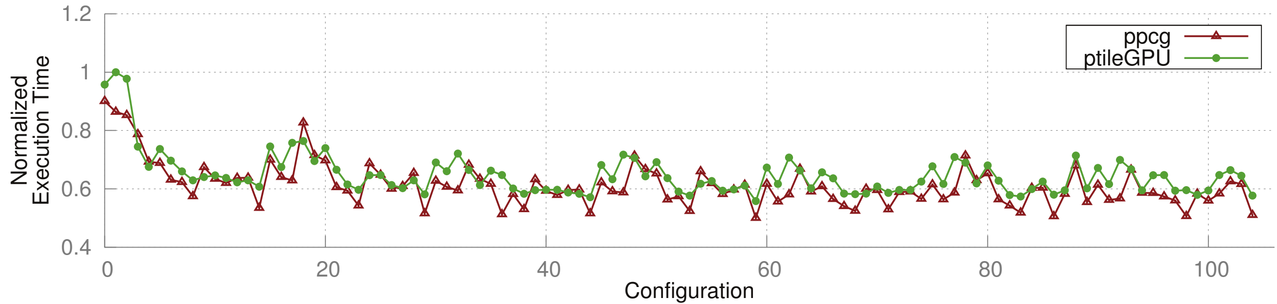
GTX580



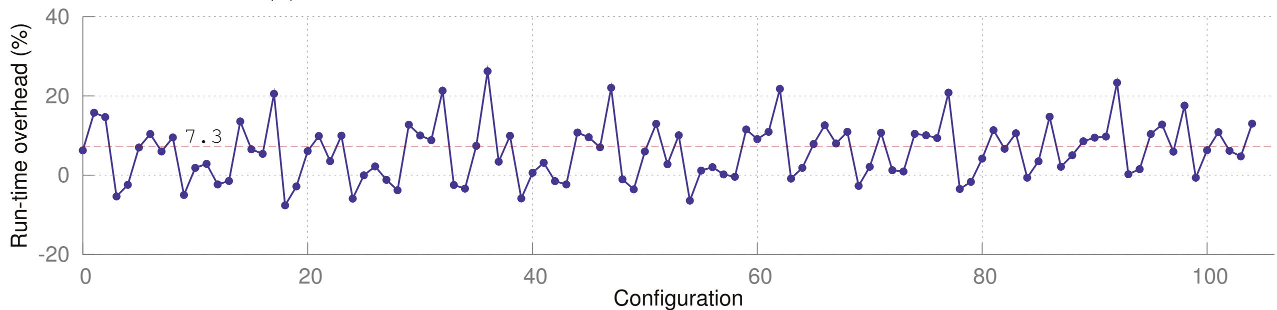
(a) No local memory. Worst execution time 0.261*sec*.



(b) Relative overhead for each configuration point of (a). Average overhead −8.4%.

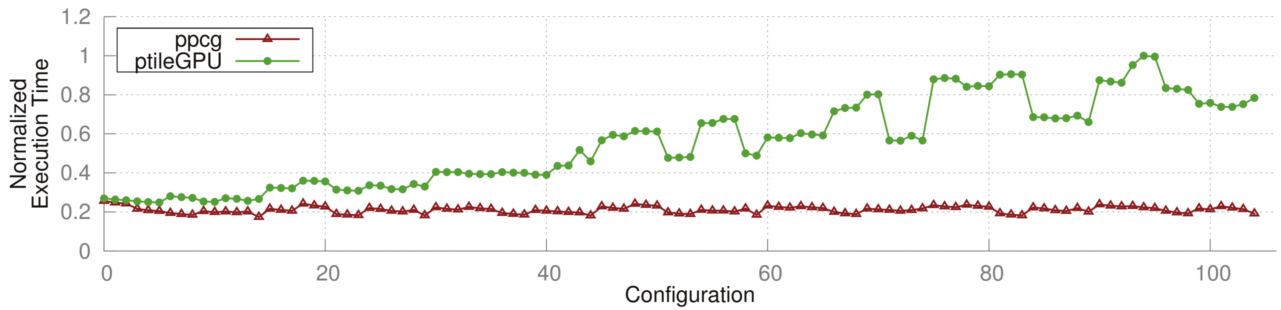


(c) With local memory. Worst execution time 0.234*sec*.

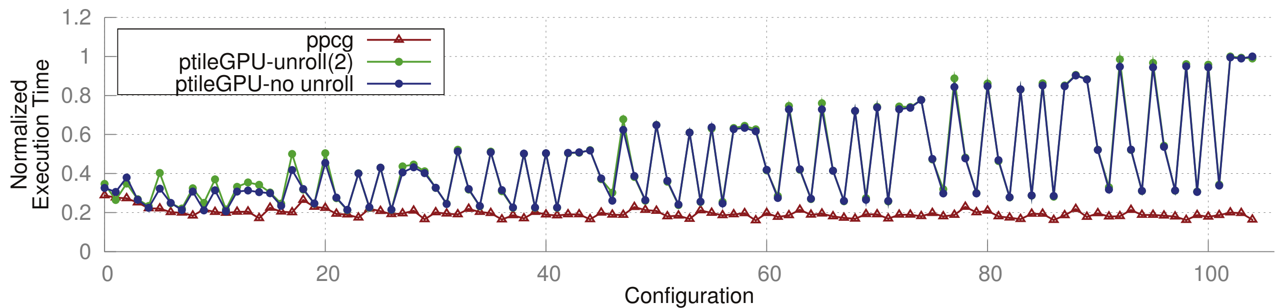


(d) Relative overhead for each configuration point of (c). Average overhead 7.3%.

Figure 6.19: Performance profiles and relative overhead of matrix-multiplication on GTX580 for a $2k \times 2k \times 2k$ problem without using thread buckets. Each point of (a) and (c) is normalized with worst execution time. The dashed red lines of (b) and (d) represent the global average run-time overhead for the entire configuration space.



(e) No local memory used. Worst execution time 0.927*sec*.



(f) With Local memory and unrolling optimization. Worst execution time 0.731*sec*.

Figure 6.20: Performance profiles of matrix-multiplication on GTX580 for a $2k \times 2k \times 2k$ problem using thread buckets. Each point of (a) and (b) is normalized with worst execution time.

M2070

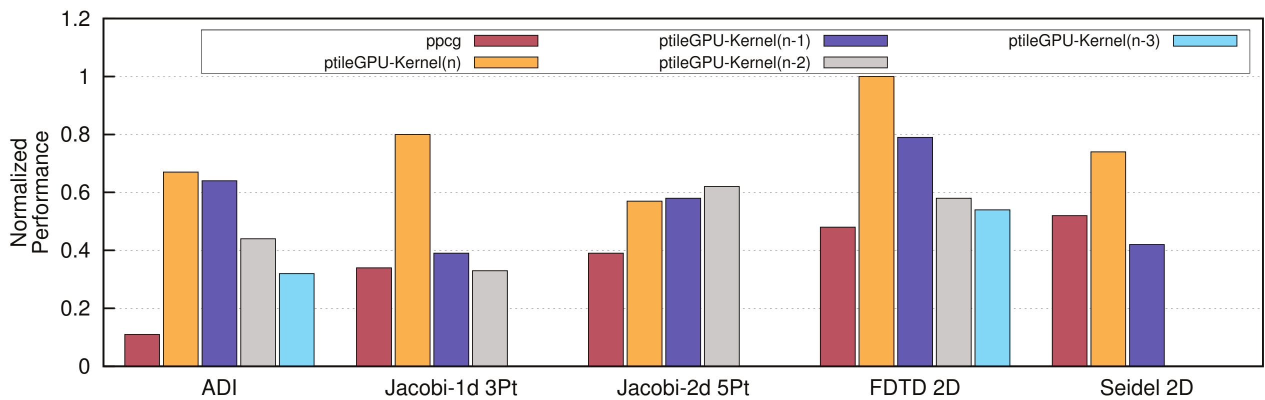
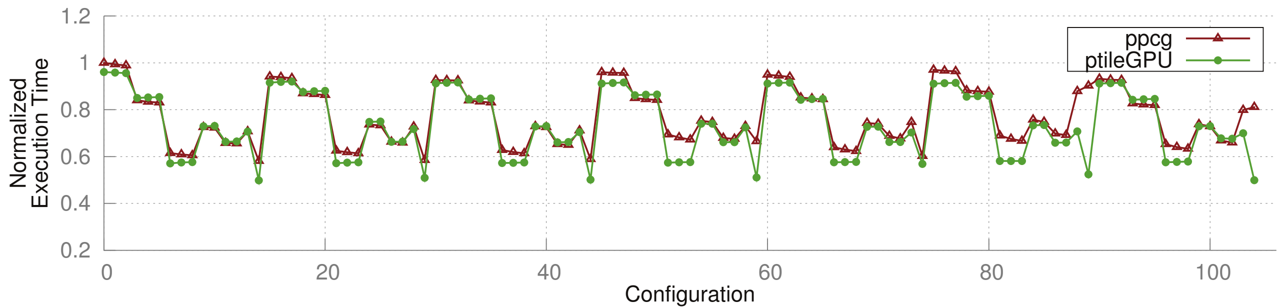
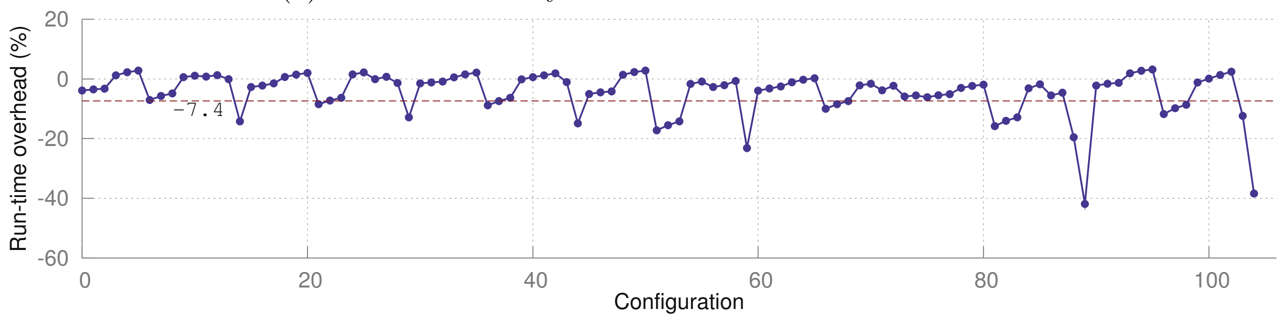


Figure 6.21: Best performances found after tile-size search on M2070. Performances are normalized with the best performance found which was 25.11 GFLOPs for Fdtd-2d.

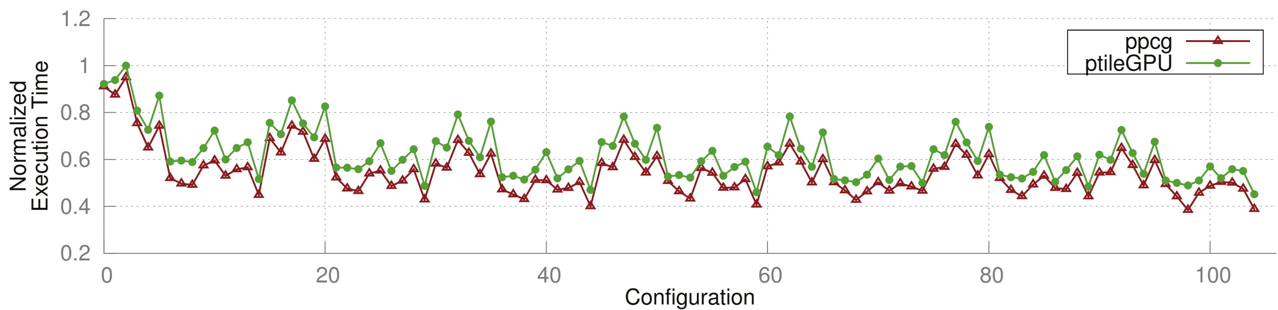
K20c



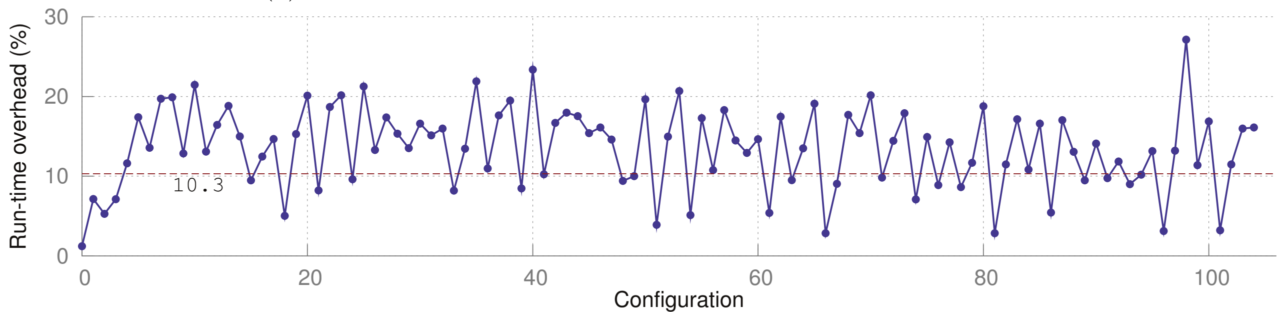
(a) No local memory. Worst execution time 0.333*sec*.



(b) Relative overhead for each configuration point of (a). Average overhead −7.4%.

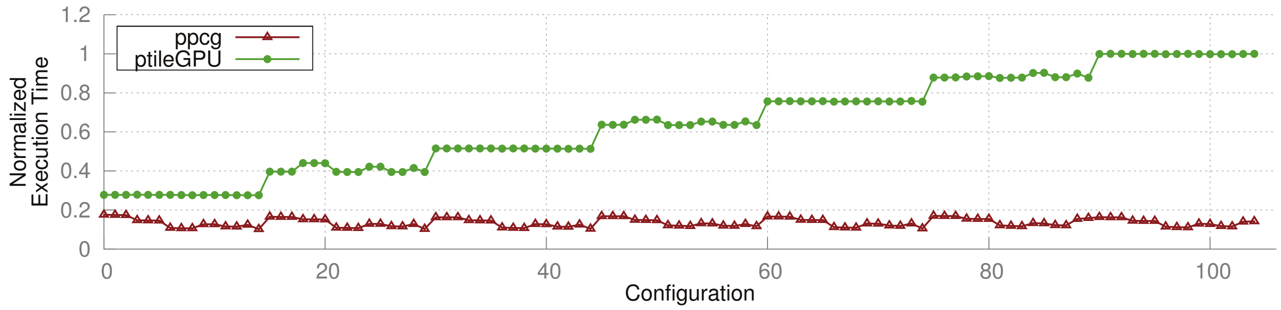


(c) With local memory. Worst execution time 0.191*sec*.

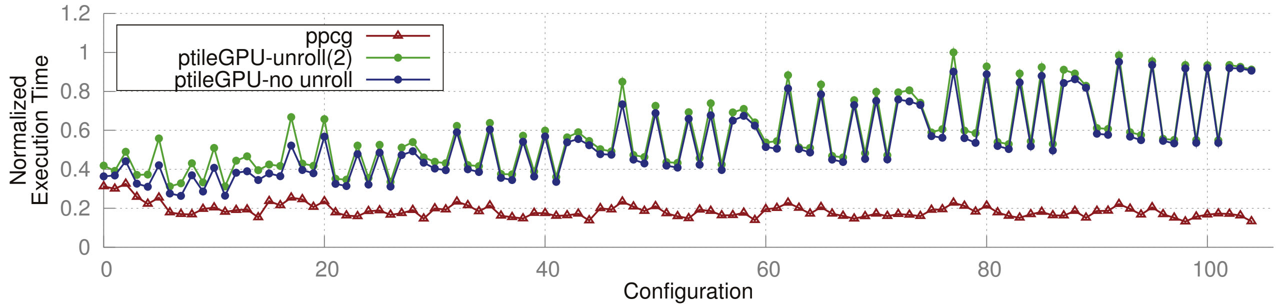


(d) Relative overhead for each configuration point of (c). Average overhead 10.3%.

Figure 6.22: Performance profiles and relative overhead of matrix-multiplication on K20c for a $2k \times 2k \times 2k$ problem without using thread buckets. Each point of (a) and (c) is normalized with worst execution time. The dashed red lines of (b) and (d) represent the global average run-time overhead for the entire configuration space.



(e) No local memory. Worst execution time 1.856*sec*.



(f) With local memory and unrolling optimization. Worst execution time 0.558*sec*.

Figure 6.23: Performance profiles of matrix-multiplication on K20c for a $2k \times 2k \times 2k$ problem using thread buckets. Each point of (a) and (b) is normalized with worst execution time.

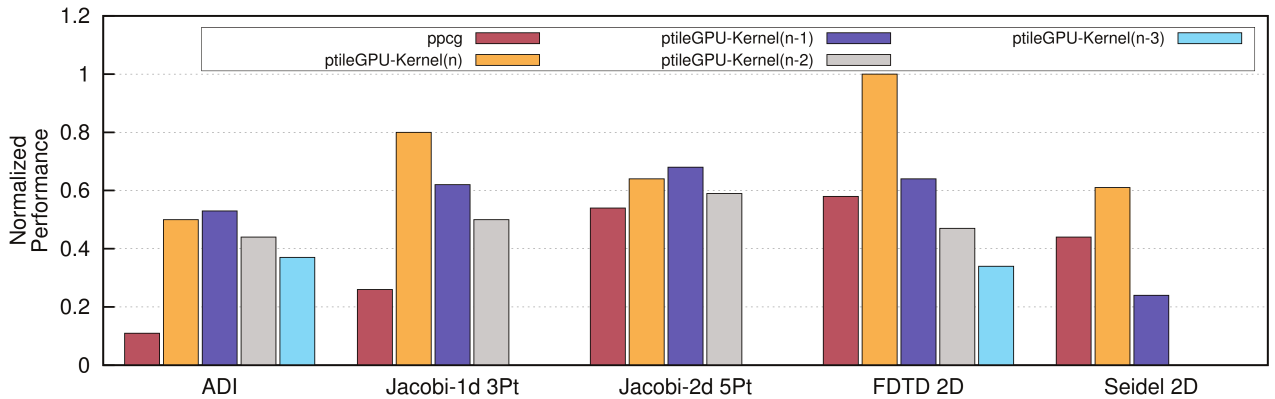


Figure 6.24: Best performances found after tile-size search on K20c. Performances are normalized with the best performance found which was 33.03 GFLOPs for Fdtd-2d.

Search times

	GTX280		GT540M		GTX580		K20c	
	ppcg	ptileGPU	ppcg	ptileGPU	ppcg	ptileGPU	ppcg	ptileGPU
matMul	*3261s*	*133s*	*8778s*	*783s*	*575s*	*95s*	*7659s*	*1237s*

Table 6.2: Tile-size search times

6.7 Conclusions

In this chapter, we presented a code generation algorithm that produces parametrically tiled GPU code for effective run-time tuning of SCoPs. The technical challenges addressed by our approach are the following :

- Extracting and mapping parametrically tiled wavefronts for GPU execution can lead to load imbalance if wavefronts are not mapped precisely to the rectangular execution space of GPUs. To address this challenge we introduced a run-time system based on the concepts of *Tile Buckets* and *Thread Buckets*.
- The parametric nature of the produced code indicates the importance of a dynamic local memory management mechanism that would be able to allocate and use local memory buffers dynamically. For that purpose we introduced the concept of *Buffer Buckets*, i.e., buckets of abstract local memory buffers that are populated dynamically by a run-time system. These buffer buckets are combined with a set of predefined parametric data-movement procedures provided in the form of API methods.

Our experimental evaluation showed that in the case of rectangularly-parallel programs (in cases where wavefront parallelism can be avoided), the produced parametrically tiled GPU code performs well enough to allow us to avoid the cost of iterative compilation for performance tuning. This means that even though our method induces run-time overhead that sometimes can be relatively high (e.g. on the GTX 280) nevertheless, it remains a good predictor of PPCG [VCJC+13] – a state-of-the-art compile-time method – as indicated by the close correlation between the lines of graphs (a) and (c). On the other hand, we showed that the proposed run-time is very effective in mapping wavefronts of parallel tiles and intra-tile points as well as managing local memory dynamically. In particular, it was shown to be clearly faster than PPCG for all 5 stencils programs we used for our second assessment.

Chapter 7

Beyond Static Control Programs : The *Avelas* Runtime System

This chapter introduces the *Avelas* runtime system, a platform-independant environment that realizes a novel execution model for manycore processors. It begins with an overview of the system (Section 7.1) followed by a description of the main technical components involved (Sections 7.2 through 7.4). Finally, a preliminary experimental evaluation of the system is presented that examines its feasibility as a general purpose execution model or as a target abstraction for source-to-source compilation. Our results show that this is a promising perspective that motivates further research.

7.1 Overview

The Avelas runtime system is an attempt to realize the theoretical concepts of tile, thread and buffer buckets that were introduced in Chapter 6 along with a set of procedures that facilitate their intuitive and platform-independent manipulation. However, it can also be viewed as an alternative execution/programming model for manycore processors that enables dynamic management of the execution environment as shown in Figure 7.1. In particular, we see that a set of dynamically managed objects is used to specify the execution space as opposed to the

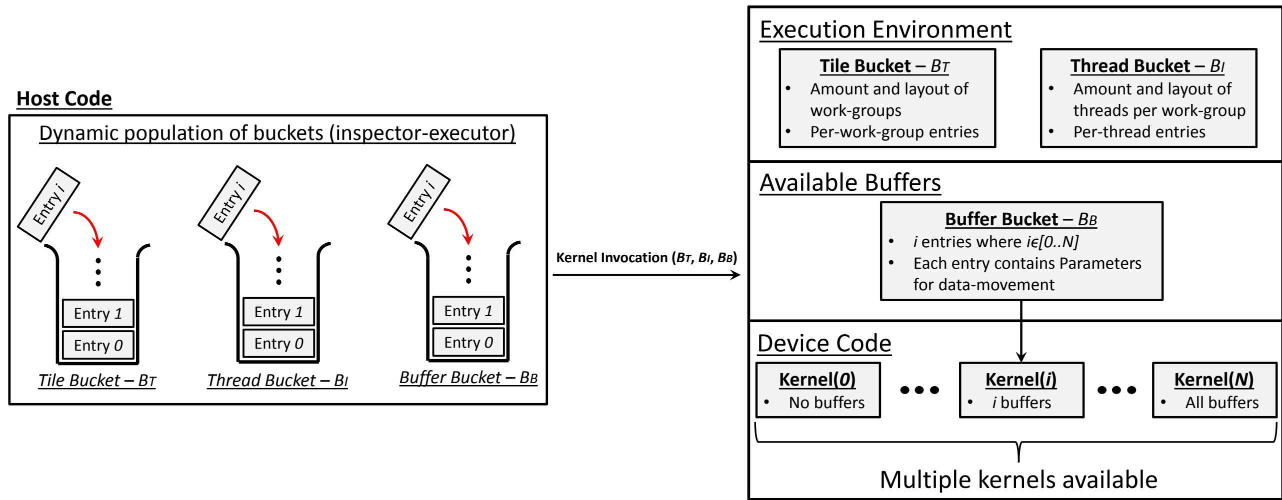


Figure 7.1: Overview of the *Avelas* execution model.

strictly rectangular configuration of the OpenCL/CUDA paradigm. These objects – the *Tile Bucket*, *Thread Bucket* and *Buffer Bucket* – contain entries that after being transferred to the device memory can provide generic index information to each work-group and work-item. In the case of buffer buckets, the entries carry information associated to specific local memory buffers and are used by data movement procedures. In Figure 7.1 we also see that buffer buckets carry a kernel descriptor as well, which is used to specify the device code to be executed by a kernel invocation (Section 6.4 describes the intuition behind this mechanism which will be detailed in Section 7.4).

In other words, the Avelas runtime system serves two main purposes. First of all, the tile bucket and thread bucket abstractions alleviate the restrictions imposed by the CUDA/OpenCL paradigm for strictly rectangular execution spaces. When the actual execution space of a program is non-rectangular (e.g. triangular or skewed), a rectangular CUDA/OpenCL mapping will lead to redundant allocation of resources in addition to the overhead involved in determining valid execution points. Secondly, the buffer bucket abstraction enables us to manage the allocation of local memory buffers dynamically from the host, which in turn facilitates runtime exploration (as opposed to compile-time exploration) of the locality/parallelism trade-off discussed in Section 6.4 and illustrated by Figure 6.8.

7.2 The *Tile Bucket*

A *Tile Bucket* corresponds to a cuda/opencl grid but without the restriction of a strictly rectangular definition. Instead, it is populated dynamically by entries, each one corresponding to a work-group and each one carrying a flexible vector of elements that uniquely identifies the owning work-group. As opposed to the built-in index variables of cuda/opencl (e.g. the `blockIdx` vector in CUDA) the elements of each tile bucket entry can carry program-specific information that is not restricted in type and are not related to a layout. In fact, this paradigm enables us to alter the layout of work-groups dynamically from the host without changing the device code. This is possible simply by altering the order in which entries are being added to the bucket. As a consequence, the device code is cleaner and the programmer doesn't need to reverse engineer the respective cuda/opencl nd-range mapping in order to retrieve the right positioning information (see example of Figure 7.4). On the other hand, by calculating work-group-wide values at the host once (instead of calculating them redundantly by each thread) and then retrieving them from the device code, we might be able to improve compute efficiency and energy consumption for the price of the bucket entry retrieval overhead. In fact, this particular trade-off will be the subject of our experimental evaluation presented in Section 7.7.

The procedures provided by the Avelas runtime to allocate and manipulate tile buckets are the following :

```
tileBucket * alloc_tile_bucket( int offset, int dims, int alloc_size=65536 );
void populate_tile_bucket( tileBucket * B, int tag, int n, ... );
void consolidate_tile_bucket( tileBucket * tb );
void free_tile_bucket( tileBucket * tb );
```

To better understand these methods lets look at the minimal example of Figure 7.2. In this example we see a bucket containing 5 entries each one having 3 elements or dimensions – the number of dimensions is specified by the `dims` and `n` arguments to the `alloc_tile_bucket` and `populate_tile_bucket` methods respectively. This setting corresponds to an nd-range consisting of 5 work-groups each one corresponding to a bucket entry. We also see that each

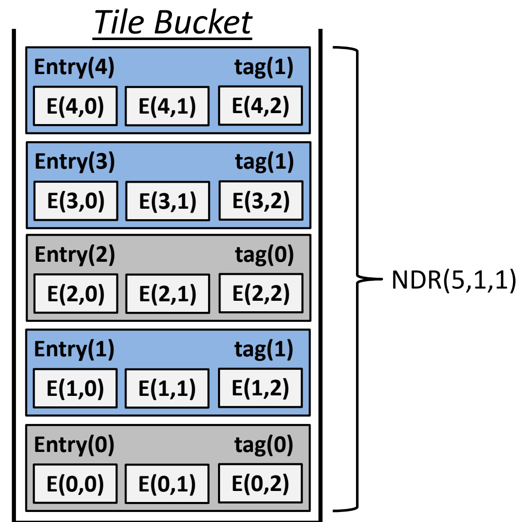


Figure 7.2: Structure of a tile bucket consisting of 5 entries each one carrying 3 elements.

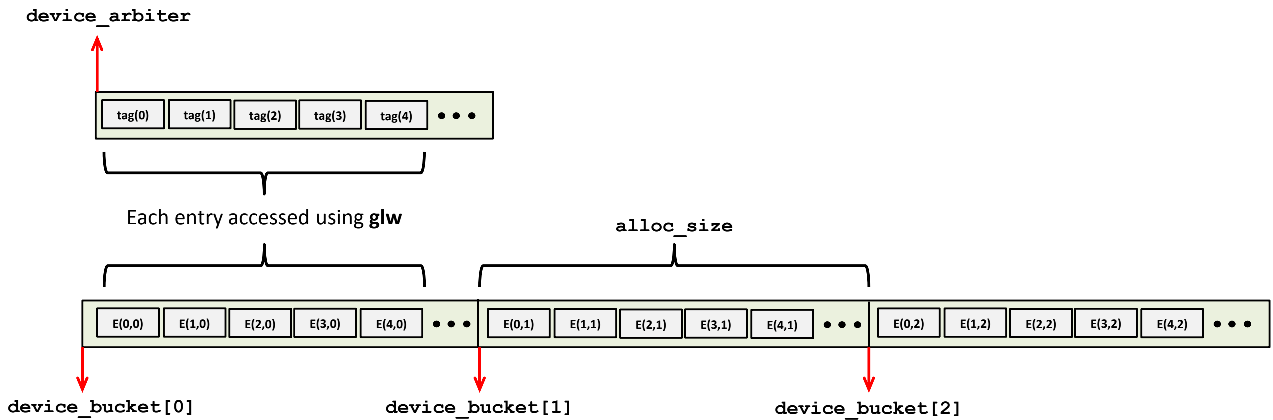


Figure 7.3: Storage layout of a tile bucket.

entry is coloured based on a `tag` attribute. This distinction can be useful in cases where we want a given set of work-groups to execute a distinct version of the computation (e.g. full tile separation) or a distinct computation altogether (e.g. different versions of an algorithm or completely different algorithms). In Figure 7.3 we see the storage layout for the tile bucket along with the device pointers provided by the `tileBucket` data structure. These pointers can be used from the device code to access the tile bucket either as a flattened 1-dimensional array (using pointer `device_bucket[0]` only) or as distinct vectors per dimension (using all `device_bucket` pointers). In the former case the `alloc_size` parameter is transferred to a constant memory position specified by the `offset` parameter and then used as the storage stride between the entry dimensions.

A `tileBucket` object is instantiated with the `alloc_tile_bucket` method which is responsible for allocating the host and device memory required to accommodate the bucket entries as well as the `arbiter` vector. After a series of invocations to `populate_tile_bucket` the `consolidate_tile_bucket` procedure is called in order to make the necessary transfers from host to device memory. Finally, all resources are released with the `free_tile_bucket` method.

The example of Figure 7.4 shows how the diagonal reordering optimization proposed by Ruetsch et al. [RM09] for matrix transpose can be implemented using tile-buckets. The `TILE_BUCKET` preprocessor macro is defined as follows:

```
TILE_BUCKET(I,VAR) __shared__ int VAR; VAR = BT_##I[glw];
```

It is clear that the device code has been greatly simplified since it has been decoupled from the `blockIdx` layout. This layout is now specified at the host code which makes it easier to debug and to alter dynamically without affecting the device code. Furthermore, several compute operations including two modulus and one division have been removed from the device code and performed once per work-group by the host.

7.3 The *Thread Bucket*

Similar to the concept of a tile bucket, a *Thread Bucket* provides an alternative abstraction to the rectangular work-group configuration of the cuda/opencl paradigm. However, unlike tile bucket entries, each thread bucket entry is not uniquely owned by each thread but shared among threads of different work-groups. Therefore each thread bucket entry uniquely identifies a thread within a work-group for each individual work-group. In the case of non-rectangular work-group spaces, such abstraction can be highly beneficial as it avoids the allocation of redundant resources (e.g. registers) as well as the control overhead associated with the rectangular overapproximation of the parallel execution space. Another useful observation has to do with the scalability of thread buckets. In particular, unlike nd-ranges, work-groups have a relatively low thread capacity limit (e.g. on an NVIDIA GPU with 2.1 compute capability, this limit

```
/* Device Code */
__global__ void transposeDiagonal(float *odata,
float *idata, int width, int height, int nreps)
{
__shared__ float tile[TILE_DIM][TILE_DIM+1];

int blockIdx_x, blockIdx_y;

// diagonal reordering
if (width == height) {
   blockIdx_y = blockIdx.x;
   blockIdx_x = (blockIdx.x+blockIdx.y)%gridDim.x;
} else {
   int bid = blockIdx.x + gridDim.x*blockIdx.y;
   blockIdx_y = bid%gridDim.y;
   blockIdx_x = ((bid/gridDim.y)+blockIdx_y)%gridDim.x;
}

int xIndex = blockIdx_x*TILE_DIM + threadIdx.x;
int yIndex = blockIdx_y*TILE_DIM + threadIdx.y;
int index_in = xIndex + (yIndex)*width;

xIndex = blockIdx_y*TILE_DIM + threadIdx.x;
yIndex = blockIdx_x*TILE_DIM + threadIdx.y;
int index_out = xIndex + (yIndex)*height;

                  /* Computation */
}
                  /* Host Code */
dim3 grid(g1,g2);
dim3 threads(t1,t2);
transposeDiagonal<<<grid,threads>>>
(d_odata, d_idata, size_x, size_y, NUM_REPS);
```

(a) Original according to [RM09]

```
                  /* Device Code */
__global__ void transposeDiagonal(float *odata,
float *idata, int width, int height, int nreps,
int * BT_0, int * BT_1)
{
__shared__ float tile[TILE_DIM][TILE_DIM+1];

TILE_BUCKET(0,blockIdx_x)
TILE_BUCKET(1,blockIdx_y)

int xIndex = blockIdx_x + threadIdx.x;
int yIndex = blockIdx_y + threadIdx.y;
int index_in = xIndex + (yIndex)*width;

xIndex = blockIdx_y + threadIdx.x;
yIndex = blockIdx_x + threadIdx.y;
int index_out = xIndex + (yIndex)*height;

        /* Computation (remains the same) */
}
                  /* Host Code */
tileBucket * BT = alloc_tile_bucket(0,2);
for ( int i = 0 ; i < g2 ; i++ ) {
   for ( int j = 0 ; i < g1 ; j++ ) {
      if ( width == height ) {
         val1 = j;   val2 = (i+j)%gridDim.x;
      } else {
         temp = j + gridDim.x*i; val1 = temp%gridDim.y;
         val2 = ((temp/gridDim.y)+val1)%gridDim.x;
      }
      populate_tile_bucket(BT,0,2,
                  val2*TILE_DIM, val1*TILE_DIM);
   }
}
consolidate_tile_bucket(BT);
dim3 threads(t1,t2);
transposeDiagonal<<<BT->config,threads>>>
(d_odata, d_idata, size_x, size_y, NUM_REPS,
BT->device_bucket[0], BT->device_bucket[1]);
```

(b) Using a tile-bucket

Figure 7.4: Diagonal reordering with and without tile-buckets.

is 1536 and 2048 for devices with 3.x compute capability) which reflects the inherent resource limitations of compute units. This indicates that thread buckets can have a bounded storage footprint that in turn enables us to utilize less scalable yet faster memory modules like constant or image memory[1]. Currently, thread buckets are stored in image memory in order to exploit the 2D locality optimizations available by the hardware.

The provided methods are the following :

```
threadBucket * alloc_thread_bucket( int id );
void set_rectangular_thread_layout( threadBucket * B, int padding, int n, ... );
int set_wavefront_thread_layout( threadBucket * B, int padding, int n, ... );
void consolidate_thread_bucket( threadBucket * tb );
void free_thread_bucket( threadBucket * tb );
```

Note that the methods for constructing a rectangular and a wavefront bucket are fixed and can be used for any program. Also the `id` argument provided to the `alloc_thread_bucket` method distinguishes between multiple `threadBucket` objects. Furthermore, the `padding` parameter is used to pad the total number of allocated threads in case we want to match the hardware's allocation granularity (e.g. 32 for NVIDIA GPUs). In both `set_rectangular_thread_layout` and `set_wavefront_thread_layout` methods, the `n` argument denotes the dimensionality of each entry or the dimensionality of the respective bounding box.

Since thread buckets are stored in image memory, the user does not need to worry about modifying the kernel arguments (this was necessary for the tile buckets) as the image memory declarations are handled automatically by the runtime and the respective objects are readily available from the device code through the following preprocessor macro :

```
THREAD_BUCKET(I,J,D,VAR)
```

In particular, we define a small 2D space of 2D image objects (currently a 4x4 space). The `I` and `J` arguments are used to specify a particular image memory object while the `D` argument

[1] Also known as texture memory.

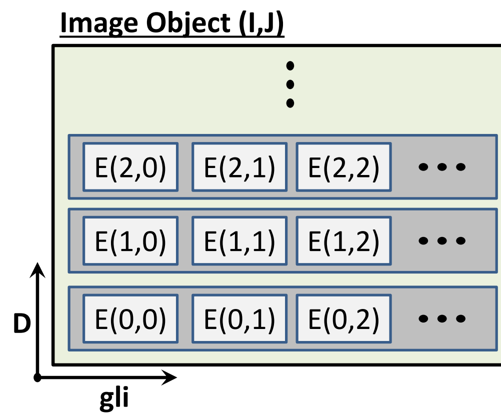


Figure 7.5: Structure of a thread bucket object residing in off-chip image memory.

is used to access the rows of a given object. For a given row of a given 2D image object the built-in **gli** index variable is used to access each column entry. Figure 7.5 depicts the storage layout for a given 2D image object.

In the case of wavefront parallelism, the `I` argument is the `id` of the respective bucket, the `J` argument specifies the dimension of the entry, `D` denotes the wavefront instance and `VAR` the name of the private variable that holds the `J` dimension of the **gli** entry of the `D` wavefront instance.

In the case of rectangularly parallel work-groups, the `I` argument is always 0, the `J` argument denotes the `id` of the thread bucket, `D` identifies the dimension of the entry and `VAR` is the name of the private variable that holds the `D` dimension of the **gli** entry of the `J` thread bucket.

7.4 The *Buffer Bucket*

In Section 6.4 we introduced the theoretical concept of a *Buffer Bucket* – a bucket of parametric buffer information that is dynamically managed according to the usage model of Figure 6.9. In this section we will discuss how buffer buckets were implemented as part of the Avelas runtime system. Unlike tile and thread bucket entries, each buffer bucket entry carries a specific list of parameters that are used by the data-movement procedures. In addition, each buffer bucket holds a kernel descriptor that identifies the device code to be executed. The purpose of this is to enable us to dynamically adjust the number of local memory buffers available to the

computation and in order to achieve that we need to have multiple versions of the device code each one corresponding to a particular set of local memory buffers.

The methods provided by the Avelas runtime system for instantiating, populating and deallocating buffer buckets are the following :

```
bufferBucket * alloc_buffer_bucket( int offset, void * kernel,
                                        int max_size, int L1_size );
void submit_buffer( int type, size_t type_size, threadBucket * BI,
                      bufferBucket * BB, bool guard, void * kernel, int dim, ... );
void consolidate_buffer_bucket( bufferBucket * bb );
void free_buffer_bucket( bufferBucket * bb );
```

In order to allocate a `bufferBucket` object we can use the `alloc_buffer_bucket` method. The `offset` parameter is similar to the one used by `alloc_tile_bucket`, i.e., distinguishes between multiple `bufferBucket` objects. The `kernel` argument should be a pointer to a device code that doesn't use any local memory buffers. The `max_size` argument initializes the local memory window or $\mathcal{L}_w$ as defined in Section 6.4. Finally, the `L1_size` argument is associated with an experimental feature that manages the partition of local memory into software managed and hardware managed cache (the ability to control this partition is only available on NVIDIA GPUs to date).

The `submit_buffer` method is essentially an implementation of Algorithm 6 with a few additional arguments. First of all, the `type` argument is used to control line 11 of Algorithm 6 and can be `STRICT` (no additional threads), `FLAT` (work-group threads are guaranteed to match the width of the buffer – this is used to eliminate the width traversal of line 6 in Algorithms 7 and 8), `HALF` (work-group threads are guaranteed to match half of the buffer – this is used to fully unroll the data-movement traversals), `FULL` (work-group threads are guaranteed to match the entire buffer – this is used to eliminate all data-movement traversals). Evidently, if `type` is not `STRICT` then the thread bucket argument `BI` will be modified accordingly. The `guard` argument is used to predicate the buffer submission method with a potentially dynamic condition (e.g. add only those buffers that have the exact same width). Finally, `kernel` is a pointer to

the respective device function while the remaining argument list consists of extent and padding factor pairs for each dimension $i \in [1..dim]$.

After a series of invocations to `submit_buffer`, the `consolidate_buffer_bucket` method is used to transfer all entries to constant memory. In particular, the Avelas runtime system maintains one constant memory vector for each entry parameter (i.e. 7 vectors for the 7 parameters added as per line 18 of Algorithm 6) and the host to device transfer starts from position `offset` and transfers `buffer_count` elements where `buffer_count` is the total number of buffer added to the bucket.

7.5 Experimental Evaluation

We now present a preliminary experimental evaluation that examines the effectiveness of the tile-bucket mechanism in substituting various nd-range configurations. Our experiments involved four CUDA benchmarks from the parboil benchmark suite [SRS+12] and are conducted on three GPU devices: the GT540M, M2070 and K20c cards as described in Table 6.1. Table 7.1 lists the four parboil benchmarks used along with a set of parameters related to the experiment. The datasets of Table 7.1 are part of the datasets provided by the parboil framework while all the benchmarks used for our experiments correspond to the `cuda_base`[2] versions.

The four benchmarks were chosen to represent different execution settings each one with a different combination of kernel invocations, work-groups per invocation and entry dimensions (i.e. elements/entry). Even though this set is not extensive, nevertheless it provides a good mix of configurations for our preliminary study. First of all, the bfs benchmark involves a large number of invocations each one using 7.3 work-groups on average. In such setting the kernel invocation cost dominates the overall execution time thus the tile-bucket mechanism adds a significant overhead without any obvious benefit. This is confirmed by the results of Table 7.2 where the cost of using tile-buckets yields more than three times slower execution time. Our second benchmark (cutcp) has a much smaller number of invocations with each invocation

[2] The parboil benchmark suite provide a number of different implementations for each benchmark including `cuda_base`, `cuda`, `opencl_base` and more.

	dataset	Kernels	invocations *(Total)*	work-groups *(Total)*	work-groups *(Average)*	elements/entry *(Average)*
bfs	1M	1	1999	14674	7.3	1
cutcp	large	1	26	70304	2704	7
histo	large	4	10000	3670000	367	2.5
mri-gridding	small	1	1	262144	262144	11

Table 7.1: List of benchmarks and the corresponding execution scenarios. All bencharks listed correspond to the respective `cuda_base` version The fifth column denotes the average number of work-groups per kernel invocation.

	GT540M		**M2070**		**K20c**	
	Avelas	*Original*	*Avelas*	*Original*	*Avelas*	*Original*
bfs	*0.44*	*0.17*	*0.52*	*0.15*	*0.57*	*0.17*
cutcp	*4.28*	*4.25*	*0.74*	*0.76*	*0.30*	*0.29*
histo	*198.6*	*200.2*	*64.1*	*63.2*	*62.0*	*61.48*
mri-gridding	*4.11*	*4.01*	*0.65*	*0.63*	*0.37*	*0.39*

Table 7.2: Average kernel execution time in seconds with less than 1% variability.

involving 2704 work-groups on average. The histo benchmark involved the largest number of kernel invocations with only 2.5 values stored per tile-bucket entry on average. Finally, the mri-gridding benchmark involved a single yet large kernel invocation with 262144 work-groups and 11 values stored per tile-bucket entry. Because the total number of work-groups for the mri-gridding benchmark exceeds the limit for a 1D nd-range configuration we used a simple factorial algorithm in order to create a 2D configuration out of the total amount of tile-bucket entries.

From Table 7.2 we see that the tile-bucket mechanism had a negligible impact on performance which in some cases was positive e.g. mri-gridding on K20c, histo on GT540M and cutcp on M2070. The bfs benchmark was an exception with the observed performance degradation attributed to the large volume of very small kernels each one executing 7.3 work-groups on average. In general, these results are encouraging because even though tile-buckets did not provide a consistent performance improvement, nevertheless they simplified the device code by decoupling it from the mapping layout. This was particularly obvious on mri-gridding as shown in Figure 7.6. Such simplification can have positive implications on programmability and debugging considering that programmers don't need to reverse engineer the mapping layout or debug complex thread-invariant calculations within the device code.

```
const int flatIdx = threadIdx.z*blockDim.y*blockDim.x+
                    threadIdx.y*blockDim.x+threadIdx.x;

// figure out starting point of the tile
const int z0 = blockDim.z*(blockIdx.y/(gridSize_c[1]/blockDim.y));
const int y0 = blockDim.y*(blockIdx.y%(gridSize_c[1]/blockDim.y));
const int x0 = blockIdx.x*blockDim.x;

const int X  = x0+threadIdx.x;
const int Y  = y0+threadIdx.y;
const int Z  = z0+threadIdx.z;

const int xl = x0-ceil(cutoff_c);
const int xL = (xl < 0) ? 0 : xl;
const int xh = x0+blockDim.x+cutoff_c;
const int xH = (xh >= gridSize_c[0]) ? gridSize_c[0]-1 : xh;

const int yl = y0-ceil(cutoff_c);
const int yL = (yl < 0) ? 0 : yl;
const int yh = y0+blockDim.y+cutoff_c;
const int yH = (yh >= gridSize_c[1]) ? gridSize_c[1]-1 : yh;

const int zl = z0-ceil(cutoff_c);
const int zL = (zl < 0) ? 0 : zl;
const int zh = z0+blockDim.z+cutoff_c;
const int zH = (zh >= gridSize_c[2]) ? gridSize_c[2]-1 : zh;

const int idx = Z*size_xy_c + Y*gridSize_c[0] + X;
```

(a)

```
TILE_BUCKET(0,t1)
TILE_BUCKET(1,t2)
TILE_BUCKET(2,z0)
TILE_BUCKET(3,y0)
TILE_BUCKET(4,x0)
TILE_BUCKET(5,xL)
TILE_BUCKET(6,xH)
TILE_BUCKET(7,yL)
TILE_BUCKET(8,yH)
TILE_BUCKET(9,zL)
TILE_BUCKET(10,zH)

const int flatIdx = threadIdx.z*t1+
                    threadIdx.y*t2+
                    threadIdx.x;

const int X  = x0+threadIdx.x;
const int Y  = y0+threadIdx.y;
const int Z  = z0+threadIdx.z;

const int idx = Z*size_xy_c + Y*gridSize_c[0] + X;
```

(b)

Figure 7.6: The section of the device code of mri-gridding affected by the use of tile-buckets. (a) Original `cuda_base` parBoil version and (b) Avelas version using tile-buckets.

7.6 Related Work

The Sequoia programming language [FHK+06] is a high-level abstraction designed to facilitate memory-hierarchy aware programming. It is based on a platform-independent runtime environment [HPR+08] that enables effective mapping of Sequoia programs to diverse memory hierarchies (e.g. distributed memory clusters and Cell BE processors). Even though a GPU implementation of the runtime is available, it is an early prototype that has not been publicly evaluated yet. One of the main differences between the Sequoia runtime and the Avelas runtime is that the former does not provide abstractions that are tunable at runtime like the bucket abstractions provided by Avelas. Instead, an abstract hardware model is provided as a template to guide the Sequoia compiler statically [RPH+08]. Furthermore, unlike Avelas, Sequoia was not designed with manycore processors in mind hence its effectiveness on GPUs remains unclear.

In addition to Sequoia, there has been a large body of industrial and academic research dedicated to raising the level of abstraction for manycore processors (Section 2.4.2). However, to the best of our knowledge none of this work enables dynamic layout and local memory management.

7.7 Conclusions

This chapter presented the Avelas runtime system; a platform-independent environment that realizes a novel execution model for manycore processors based on the tile-, thread- and buffer bucket concepts introduced in Chapter 6. A preliminary experimental evaluation showed that the tile-bucket mechanism provides a promising abstraction that facilitates simpler device code (that is also easier to debug) and enables dynamic management of the work-group layouts.

Chapter 8

Conclusions

This thesis examined the issue of performance portability for manycore processors (GPUs), i.e. the ability of software to perform well on a variety of GPU devices. Even though high-level languages like OpenCL and CUDA allow us to unlock the computing power of GPUs, they are still inherently exposed to machine-dependent resource trade-offs. As a result, getting the full potential out of our hardware requires careful and explicit management of resources that is not guaranteed to work well on every device. Source-to-source compilation can help us mitigate this problem by realizing higher levels of abstraction mapped to the lower-level OpenCL or CUDA languages. By doing that we are able to automate the explicit management of resources exposed to OpenCL and CUDA and therefore attain performance portability.

First of all, in order to address the issue of performance portability an important decision needs to be made. In particular, a certain domain of programs needs to be chosen, for which performance portability can be a tractable and well-defined problem. Proposing a method that works well for every program is an unrealistic task simply because it implies that every possible program behavior can be understood equally well by the same unified theory. Even if this becomes a realistic scenario sometime in the future, our responsibility today is to understand individual classes of programs reasonably well in order to figure out how we can then move towards a more general approach. That said, our study was focused on a class of loop-programs, i.e. Static Control Programs or SCoPs, for which we can leverage elegant mathematical abstractions and

tools for analysis and optimization (i.e. the polyhedral model). Even though SCoPs represent a restrictive set of programs, we believe that it is nonetheless an important and large enough set to deserve our attention.

Once we have decided about the class of programs we are going to concentrate on, we then need to set out a strategy. In practice, there are two main strategies towards performance portability regardless of the class of programs we choose to focus on. We can either rely on a single sophisticated compilation step (static or JIT) that would be responsible for producing the right code for a given device, or compile our program multiple times (i.e. iterative compilation) and pick the optimal instance for our hardware – a process widely known as *auto-tuning* (a combination of both methods is possible too). In this thesis we investigate a third approach. In particular, we showed that for a certain class of loop-programs, i.e. SCoPs, we can combine a single compilation step – for machine-independent automatic parallelization and parametric tiling – with a run-time system that allows run-time tuning of our parametrically tiled loop-program thus avoiding the cost of iterative compilation. Consequently, we claim that performance portability is attainable for SCoPs at low cost, by utilizing parametric tiling and a novel run-time system in order to decouple the compiler from the performance tuning process. Our claim is supported by an experimental study showing that our source-to-source compilation method and run-time system can match the performance of a state-of-the-art compile-time method (with no run-time support) to a satisfactory degree and can also yield significant performance speed-ups for wavefront-parallel SCoPs.

Furthermore, it was shown that the run-time system we developed (called Avelas) can actually be effective as a general purpose programming model as well. A preliminary experimental study showed that Avelas yields very low overhead when used to implement a set of four ParBoil benchmarks while offering surprising benefits in terms of programmability and debugging. In particular, by decoupling the thread layout from the device code using tile bukets, Avelas offers a very convenient programming abstraction where the programmer does not need to reverse engineer the data layout from the thread layout. Instead, the mapping of threads to data takes place at the host which is easier to understand and debug since it is a sequential single-threaded operation.

However, the evaluation of Avelas presented in this thesis was only a first step. In order to derive more definitive conclusions we would need to do a more extensive evaluation of the system, which constitutes one of our primary objectives for the future. More specifically, we would like to investigate the effectiveness of the thread-bucket and buffer-bucket mechanisms in implementing a variety of GPU applications. These applications might include large-scale high-performance computing applications with irregular control behavior and access patterns. In addition, we would like to evaluate alternative implementation strategies for all three bucket mechanisms like the use of constant memory for storing the thread buckets or the use of special hardware optimizations wherever possible. We also believe that Avelas can be a concrete first step towards a new programming language for GPUs that would be low-level enough to exploit the full potential of GPUs while at the same time high-level enough to not be tied to specific machine-dependent and/or platform-dependent implementations. Implementing this language would always yield a parameterized code amenable to run-time tuning.

Finally, one of the fundamental properties of SCoPs is that they can be analyzed and optimized using the polyhedral model. This allows us to apply powerful automatic parallelization and tiling techniques, like the Pluto scheduling algorithm, and use well-known mathematical tools – like ILP solvers, linear programming etc – to reason about memory access patterns and data movement. Exploiting these technologies though requires a practical polyhedral framework. This thesis introduced such a framework, RosePolly, an object-oriented API for polyhedral compilation. The main motivation behind the development of RosePolly was to develop a modular and flexible API that would allow us to customize existing polyhedral technologies like Pluto and CLooG and also build custom stand-alone tools easily. This was proven a valuable asset for our primary research objectives as it helped us implement key algorithms from Chapter 6 (i.e. Algorithms 4 and 6.4.2). However, the early development stage of RosePolly did not allow us to implement the complete compilation flow proposed in Chapter 6. We believe that a complete implementation would have helped us crystalize the essential requirements for our system and therefore provide a simpler and more easily understandable Chapter 6. In addition, an extensive experimental evaluation across the entire polybench suite would have been possible and would have provided stronger evidence for our position.

8.1 Future Research and Development

Here are a few pragmatic suggestions for further research and development:

- Completion of the development and testing of RosePolly.
- A completed RosePolly framework would allow us to implement the entire compilation flow proposed in Chapter 6.
- Utilize the completed compiler to apply the proposed scheme on the entire polyBench suite.
- Development of an OpenCL implementation for the Avelas run-time system that would enable a more extensive experimental evaluation across a larger set of devices.
- Investigate alternative implementation choices for the Avelas run-time.
- Formulation of GPU performance models that would drive a run-time auto-tuning system.
- Development and evaluation of a new programming language for GPUs that would be based on the Avelas system and produce parameterized code amenable to run-time performance tuning.

Bibliography

[ABC+06] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006. 9

[ABCR10] Joshua Auerbach, David F Bacon, Perry Cheng, and Rodric Rabbah. Lime: a java-compatible and synthesizable language for heterogeneous architectures. *ACM Sigplan Notices*, 45(10):89–108, 2010. 26

[ACE+12] Mehdi Amini, Béatrice Creusillet, Stéphanie Even, Ronan Keryell, Onig Goubier, Serge Guelton, Janice Onanian McMahon, François-Xavier Pasquier, Grégoire Péan, Pierre Villalon, et al. Par4all: From convex array regions to heterogeneous computing. In *IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012*, 2012. 24, 56, 71

[AHKB00] Vikas Agarwal, MS Hrishikesh, Stephen W Keckler, and Doug Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 248–259. ACM, 2000. 8

[AK87] Randy Allen and Ken Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(4):491–542, 1987. 81

[ALSU07] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: principles, techniques, and tools*, volume 1009. Pearson/Addison Wesley, 2007. 23, 33, 38, 91

[Bas04] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16. IEEE Computer Society, 2004. 34, 80, 88

[BBK+08a] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. A compiler framework for optimization of affine loop nests for GPGPUs. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 225–234. ACM, 2008. 24, 25, 38

[BBK+08b] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Compiler Construction*, volume 4959 of *Lecture Notes in Computer Science*, pages 132–146. Springer Berlin / Heidelberg, 2008. 24, 30, 34, 38, 39, 44, 53, 70, 83, 102

[BBL+12] C Bertolli, A Betts, N Loriant, G Mudalige, D Radford, D Ham, MB Giles, and P Kelly. Compiler optimizations for industrial unstructured mesh cfd applications on gpus. In *at Languages and Compilers for Parallel Computing Workshop*, 2012. 28

[BCK11] Michael Bauer, Henry Cook, and Brucek Khailany. CudaDMA: optimizing GPU memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 12. ACM, 2011. 93

[BDG+04] Jairo Balart, Alejandro Duran, Marc Gonzàlez, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. Nanos mercurium: a research compiler for openmp. In *Proceedings of the European Workshop on OpenMP*, volume 8, 2004. 22

[BDP+10] Sara S Baghsorkhi, Matthieu Delahaye, Sanjay J Patel, William D Gropp, and Wen-mei W Hwu. An adaptive performance modeling tool for GPU architectures. *ACM Sigplan Notices*, 45(5):105–114, 2010. 18

[BF03] Cédric Bastoul and Paul Feautrier. Improving data locality by chunking. In *Compiler Construction*, pages 320–334. Springer, 2003. 69

[BH11] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for CUDA. *GPU Computing Gems: Jade Edition*, pages 359–372, 2011. 26

[BHT+10] Muthu Manikandan Baskaran, Albert Hartono, Sanket Tavarageri, Thomas Henretty, J Ramanujam, and P Sadayappan. Parameterized tiling revisited. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 200–209. ACM, 2010. 70, 73, 75, 77, 78, 79

[Bon98] Probir K Bondyopadhyay. Moore's law governs the silicon revolution. *Proceedings of the IEEE*, 86(1):78–81, 1998. 7

[BR07] Uday Bondhugula and J Ramanujam. Pluto: A practical and fully automatic polyhedral parallelizer and locality optimizer. 2007. 24, 34, 38, 39, 44, 53, 55, 70, 102

[BRS10] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Compiler Construction*, pages 244–263. Springer, 2010. 24, 71

[BTSA12] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012. 27

[BYF+09] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174. IEEE, 2009. 18

[CBM+09] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE, 2009. 18, 19

[CHA12] Linchuan Chen, Xin Huo, and Gagan Agrawal. Accelerating MapReduce on a coupled CPU-GPU architecture. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 25. IEEE Computer Society Press, 2012. 27

[Che12] Chun Chen. Polyhedra scanning revisited. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 499–508. ACM, 2012. 80, 88

[CLdM12] Luke Cartey, Rune Lyngsoe, and Oege de Moor. Synthesising graphics card programs from DSLs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 121–132. ACM, 2012. 28

[CSB92] Anantha P Chandrakasan, Samuel Sheng, and Robert W Brodersen. Low-power CMOS digital design. *IEICE TRANSACTIONS on Electronics*, 75(4):371–382, 1992. 9

[CZZ12] Jason Cong, Peng Zhang, and Yi Zou. Optimizing memory hierarchy allocation with loop transformations for high-level synthesis. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1233–1238. ACM, 2012. 38

[DBB07] Romain Dolbeau, Stéphane Bihan, and François Bodin. HMPP: A hybrid multi-core parallel programming environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007. 28

[DCR+12] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F Bacon, and Stephen J Fink. Compiling a high-level language for GPUs:(via language support for architectures and compilers). In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 1–12. ACM, 2012. 26

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. 27

[DGR+74] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, 1974. 7

[DMV+08] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 4. IEEE Press, 2008. 28

[DR96] Michèle Dion and Yves Robert. Mapping affine loop nests. *Parallel Computing*, 22(10):1373–1397, 1996. 37

[DRVV00] Alain Darte, Yves P Robert, Frederic Vivien, and Frédéric Vivien. *Scheduling and automatic Parallelization*. Springer, 2000. 38

[DV94] Alain Darte and Frédéric Vivien. Automatic parallelization based on multi-dimensional scheduling. *Res. Report*, (94-24), 1994. 37

[DX11] Peng Di and Jingling Xue. Model-driven tile size selection for DOACROSS loops on GPUs. In *Euro-Par 2011 Parallel Processing*, pages 401–412. Springer, 2011. 24, 71

[Fea88] Paul Feautrier. Parametric integer programming. *RAIRO Recherche opérationnelle*, 22(3):243–268, 1988. 47

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991. 41

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem. i. one-dimensional time. *International journal of parallel programming*, 21(5):313–347, 1992. 46

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem. part II. multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992. 23, 37, 56

[Fea94] Paul Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994. 37

[FHK+06] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J Dally, et al. Sequoia: programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83. ACM, 2006. 130

[FJ07] Massimiliano Fatica and Won-Ki Jeong. Accelerating matlab with cuda. In *The High Performance Embedded Computing Workshop*, 2007. 27

[FM+11] Samuel H Fuller, Lynette I Millett, et al. *The Future of Computing Performance: Game Over or Next Level?* National Academies Press, 2011. 7, 8, 9, 10

[FOW87] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987. 18

[GCK+13] Tobias Grosser, Albert Cohen, Paul HJ Kelly, J Ramanujam, P Sadayappan, and Sven Verdoolaege. Split tiling for GPUs: automatic parallelization using

trapezoidal tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 24–31. ACM, 2013. 24, 71, 108

[GGHVDG01] John A Gunnels, Fred G Gustavson, Greg M Henry, and Robert A Van De Geijn. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software (TOMS)*, 27(4):422–455, 2001. 28

[GGL12] TOBIAS GROSSER, ARMIN GROESSLINGER, and CHRISTIAN LENGAUER. Pollyperforming polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04), 2012. 34

[GHK+11] Benedict Gaster, Lee Howes, David R Kaeli, Perhaad Mistry, and Dana Schaa. Heterogeneous computing with OpenCL. 2011. 12

[GKZ12] Michael Garland, Manjunath Kudlur, and Yili Zheng. Designing a unified programming model for heterogeneous machines. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012. 26, 29

[Gri04] Martin Griebl. *Automatic parallelization of loop programs for distributed memory architectures*. Univ. Passau, 2004. 37, 41, 61, 69, 70

[Grö09] A. Größlinger. Precise management of scratchpad memories for localising array accesses in scientific codes. In *Compiler Construction*, pages 236–250. Springer, 2009. 24, 108

[GSBS11] Max Grossman, Alina Simion Sbîrlea, Zoran Budimlić, and Vivek Sarkar. CnC-CUDA: declarative programming for GPUs. In *Languages and Compilers for Parallel Computing*, pages 230–245. Springer, 2011. 28

[GVB+06] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, 2006. 34, 39

[HA09] T.D. Han and T.S. Abdelrahman. hi CUDA: a high-level directive-based language for GPU programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 52–61. ACM, 2009. 26, 28

[HBB+09] Albert Hartono, Muthu Manikandan Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, J Ramanujam, and Ponnuswamy Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *Proceedings of the 23rd international conference on Supercomputing*, pages 147–157. ACM, 2009. 69, 70

[HBRS10] Albert Hartono, Muthu Manikandan Baskaran, J Ramanujam, and P Sadayappan. DynTile: Parametric tiled loop generation for parallel execution on multicore processors. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010. 70

[HCC+10] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. MapCG: writing parallel program portable between CPU and GPU. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 217–226. ACM, 2010. 27, 29

[HD04] Mark Horowitz and William Dally. How scaling will change processor architecture. In *Solid-State Circuits Conference, 2004. Digest of Technical Papers. ISSCC. 2004 IEEE International*, pages 132–133. IEEE, 2004. 8

[HFL+08] B. He, W. Fang, Q. Luo, N.K. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM, 2008. 27

[HK09] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. *ACM SIGARCH Computer Architecture News*, 37(3):152–163, 2009. 16, 17

[HP11] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2011. 7, 8

[HPR+08] Mike Houston, Ji-Young Park, Manman Ren, Timothy Knight, Kayvon Fatahalian, Alex Aiken, William Dally, and Pat Hanrahan. A portable runtime interface for multi-level memory hierarchies. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 143–152. ACM, 2008. 29, 130

[HPS12] J. Holewinski, L.N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on GPU architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 311–320. ACM, 2012. 28, 29, 71, 108

[IT88] François Irigoin and Remi Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–329. ACM, 1988. 69

[KA01] Ken Kennedy and John R Allen. Optimizing compilers for modern architectures: a dependence-based approach. 2001. 39

[KBB+07] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J Ramanujam, Atanas Rountev, and P Sadayappan. Effective automatic parallelization of stencil computations. In *ACM Sigplan Notices*, volume 42, pages 235–244. ACM, 2007. 28, 70, 108

[KPL+09] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, Ahmed Fasih, AD Sarma, D Nanongkai, G Pandurangan, P Tetali, et al. PyCUDA: GPU run-time code generation for high-performance computing. *Arxiv preprint*, 2009. 27

[KR] DaeGon Kim and Sanjay Rajopadhye. Parameterized tiling for imperfectly nested loops. 69

[KRR+07] DaeGon Kim, Lakshminarayanan Renganarayanan, Dave Rostron, Sanjay Rajopadhye, and Michelle Mills Strout. Multi-level tiling: M for the price of one. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 51. ACM, 2007. 69

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004. 20

[LCL99] Amy W Lim, Gerald I Cheong, and Monica S Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th international conference on Supercomputing*, pages 228–237. ACM, 1999. 38, 55

[LE10] Seyong Lee and Rudolf Eigenmann. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010. 27, 28

[Len93] Christian Lengauer. Loop parallelization in the polytope model. In *CONCUR'93*, pages 398–416. Springer, 1993. 55

[LJE04] Sang-Ik Lee, Troy A Johnson, and Rudolf Eigenmann. Cetus–an extensible compiler infrastructure for source-to-source transformation. In *Languages and Compilers for Parallel Computing*, pages 539–553. Springer, 2004. 22

[LL98] Amy W Lim and Monica S Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel computing*, 24(3-4):445–475, 1998. 38

[LLL01] Amy W Lim, Shih-Wei Liao, and Monica S Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *ACM SIGPLAN Notices*, volume 36, pages 103–112. ACM, 2001. 69

[LME09] S. Lee, S.J. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. *ACM Sigplan Notices*, 44(4):101–110, 2009. 27

[LP93] Wei Li and Keshav Pingali. *A singular loop transformation framework based on non-singular matrices*. Springer, 1993. 47

[LRW91] Monica D Lam, Edward E Rothberg, and Michael E Wolf. The cache performance and optimizations of blocked algorithms. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 63–74. ACM, 1991. 69

[LV12] Seyong Lee and Jeffrey S Vetter. Early evaluation of directive-based gpu programming models for productive exascale computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 23. IEEE Computer Society Press, 2012. 28

[LVM+10] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 51–61. ACM, 2010. 24, 28, 56

[MGR+12] GR Mudalige, MB Giles, I Reguly, C Bertolli, and PHJ Kelly. OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In *Innovative Parallel Computing (InPar), 2012*, pages 1–12. IEEE, 2012. 28

[MM10] Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled GPU functions in haskell. In *ACM Sigplan Notices*, volume 45, pages 67–78. ACM, 2010. 28

[Moo65] Gordon E. Moore. Cramming more components into integrated circuits. *Electronics Magazine*, 1965. 7

[Moo11] Chuck Moore. Data processing in exascale-class computer systems. Technical report, The Salishan Conference on High Speed Computing, 2011. 9

[MS09] Jiayuan Meng and Kevin Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proceedings of the 23rd international conference on Supercomputing*, pages 256–265. ACM, 2009. 28, 71

[Nvi11] CUDA Nvidia. NVIDIA CUDA programming guide, 2011. 17, 18

[ONH+96] Kunle Olukotun, Basem A Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *ACM Sigplan Notices*, volume 31, pages 2–11. ACM, 1996. 9

[PBB] Louis-Noel Pouchet, Cédric Bastoul, and Uday Bondhugula. PoCC: the polyhedral compiler collection, 2010. 21

[PBB+10] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J Ramanujam, and P Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11. IEEE, 2010. 39, 56

[PBB+11] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J Ramanujam, P Sadayappan, and Nicolas Vasilache. Loop transformations: convexity, pruning and optimization. In *ACM SIGPLAN Notices*, volume 46, pages 549–562. ACM, 2011. 39

[Pen55] Roger Penrose. A generalized inverse for matrices. In *Proc. Cambridge Philos. Soc*, volume 51, pages 406–413. Cambridge Univ Press, 1955. 47

[PH09] D.A. Patterson and J.L. Hennessy. *Computer organization and design: the hardware/software interface.* Morgan Kaufmann, 2009. 8

[PW94] William Pugh and David Wonnacott. *An exact method for analysis of value-based array data dependences.* Springer, 1994. 41, 61

[PW95] William Pugh and David Wonnacott. Going beyond integer programming with the Omega test to eliminate false data dependences. *Parallel and Distributed Systems, IEEE Transactions on*, 6(2):204–211, 1995. 41, 61

[QRW00] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, 2000. 43, 55, 80, 88

[Qui00] Dan Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000. 21

[Ram95] Jagannathan Ramanujam. Beyond unimodular transformations. *The Journal of Supercomputing*, 9(4):365–389, 1995. 34

[RKRS07] Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay Rajopadhye, and Michelle Mills Strout. Parameterized tiled loops for free. *ACM SIGPLAN Notices*, 42(6):405–414, 2007. 69

[RM09] Greg Ruetsch and Paulius Micikevicius. Optimizing matrix transpose in CUDA. *Nvidia CUDA SDK Application Note*, 2009. 18, 86, 122, 123

[RPH+08] Manman Ren, Ji Young Park, Mike Houston, Alex Aiken, and William J Dally. A tuning framework for software-managed memory hierarchies. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 280–291. ACM, 2008. 130

[RRB+08] Shane Ryoo, Christopher I Rodrigues, Sara S Baghsorkhi, Sam S Stone, David B Kirk, and Wen-mei W Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82. ACM, 2008. 16, 17

[RRP+07] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor

systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13–24. IEEE, 2007. 27

[SCF+05] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, Jonathan Freeman, and Barbara Kreaseck. Combining performance aspects of irregular gauss-seidel via sparse tiling. In *Languages and Compilers for Parallel Computing*, pages 90–110. Springer, 2005. 70

[Sch98] Alexander Schrijver. *Theory of linear and integer programming.* Wiley, 1998. 45

[SCS10] Joel Svensson, Koen Claessen, and Mary Sheeran. GPGPU kernel implementation and refinement using obsidian. *Procedia Computer Science*, 1(1):2065–2074, 2010. 28

[SDKV12] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. A performance analysis framework for identifying potential benefits in GPGPU applications. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 11–22. ACM, 2012. 16, 17, 18

[SRS+12] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and WMW Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012. 127

[TCE+10] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, Ramakrishna Upadrasta, et al. Graphite two years after: First lessons learned from real-world polyhedral compilation. In *GCC Research Opportunities Workshop (GROW'10)*, 2010. 56

[UCB11] Didem Unat, Xing Cai, and Scott B Baden. Mint: realizing CUDA performance in 3D stencil methods with annotated c. In *Proceedings of the international conference on Supercomputing*, pages 214–224. ACM, 2011. 28

[ULBH08] S.Z. Ueng, M. Lathara, S. Baghsorkhi, and W. Hwu. CUDA-lite: Reducing GPU programming complexity. *Languages and Compilers for Parallel Computing*, pages 1–15, 2008. 26, 38

[VBCG06] Nicolas Vasilache, Cédric Bastoul, Albert Cohen, and Sylvain Girbal. Violated dependence analysis. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 335–344. ACM, 2006. 39

[VCJC+13] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):54, 2013. 24, 56, 71, 102, 117

[VDKV00] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000. 28

[Ver10] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *Mathematical Software–ICMS 2010*, pages 299–302. Springer, 2010. 56

[Vol10] V. Volkov. Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference, GTC*, volume 10, 2010. 16, 17

[WFW+94] Robert P Wilson, Robert S French, Christopher S Wilson, Saman P Amarasinghe, Jennifer M Anderson, Steve WK Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W Hall, Monica S Lam, et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM Sigplan Notices*, 29(12):31–37, 1994. 21

[WL91] Michael E Wolf and Monica S Lam. A data locality optimizing algorithm. *ACM Sigplan Notices*, 26(6):30–44, 1991. 69, 73

[WM95] Wm A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995. 8

[Wol86] Michael Wolfe. Loops skewing: The wavefront method revisited. *International Journal of Parallel Programming*, 15(4):279–293, 1986. 69, 74

[Wol89] Michael Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664. ACM, 1989. 69

[Wol90] Michael Joseph Wolfe. *Optimizing supercompilers for supercomputers*. MIT press, 1990. 39

[WPSAM10] H. Wong, M.M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 235–246. IEEE, 2010. 17

[WSTaM12] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACCFirst experiences with real-world applications. In *Euro-Par 2012 Parallel Processing*, pages 859–870. Springer, 2012. 28

[Xue00] Jingling Xue. *Loop tiling for parallelism*. Springer, 2000. 69

[YXKZ10] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU compiler for memory optimization and parallelism management. In *ACM Sigplan Notices*, volume 45, pages 86–97. ACM, 2010. 25, 26, 69, 102

[YXM+12] Yi Yang, Ping Xiang, Mike Mantor, Norm Rubin, and Huiyang Zhou. Shared memory multiplexing: a novel way to improve GPGPU throughput. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 283–292. ACM, 2012. 15