© 2017 Li-Wen Chang

TOWARD PERFORMANCE PORTABILITY FOR CPUS AND GPUS THROUGH ALGORITHMIC COMPOSITIONS

BY

LI-WEN CHANG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Doctoral Committee:

Professor Wen-mei W. Hwu, Chair
Professor Deming Chen
Associate Professor Nam Sung Kim
Associate Professor Steven S. Lumetta

ABSTRACT

The diversity of microarchitecture designs in heterogeneous computing systems allows programs to achieve high performance and energy efficiency, but results in substantial software redevelopment cost for each type or generation of hardware. To mitigate this cost, a performance portable programming system is required.

This work presents my solution to the performance portability problem. I argue that a new language is required for replacing the current practices of programming systems to achieve practical performance portability. To support my argument, I first demonstrate the limited performance portability of the current practices by showing quantitative and qualitative evidences. I identify the main limiting issues of conventional programming languages. To overcome the issues, I propose a new modular, composition-based programming language that can effectively express an algorithmic design space with functional polymorphism, and a compiler that can effectively explore the design space and facilitate many high-level optimization techniques. This proposed approach achieves no less than 70% of the performance of highly optimized vendor libraries such as Intel MKL and NVIDIA CUBLAS/CUSPARSE on an Intel i7-3820 Sandy Bridge CPU, an NVIDIA C2050 Fermi GPU, and an NVIDIA K20c Kepler GPU.

To my family, for their love and support.

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Wen-mei W. Hwu, for his tremendous mentorship, patience and support. He has always motivated my work and been patient with me. His wisdom will keep inspiring me in my professional career and personal life.

I would like to thank all members of the IMPACT research group, past and present, for their help and camaraderie. They are, in no particular order, Chris Rodrigues, Sara Baghsorkhi, Alex Papakonstantinou, John Stratton, I-Jui Sung, Xiao-Long Wu, Nady Obeid, Victor Huang, Deepthi Nandakumar, Hee-Seok Kim, Nasser Anssari, Tim Wentz, Daniel Liu, Izzat El Hajj, Steven Wu, Abdul Dakkak, Simon Garcia de Gonzalo, Wei-Sheng Huang, Carl Pearson, Cheng Li, Sitao Huang, Tom Jablin, John Larson, Chia-Jen Chang, Chih-Sheng Lin, Judit Planas, Jose Cecilia, Juan Gómez Luna, Javier Cabezas, Isaac Gelado, Nacho Navarro, Hiroyuki Takizawa, Xuhao Chen, Omer Anjum, and Mohamed El Hadedy.

Particularly, I would like to thank Izzat El Hajj, Simon Garcia de Gonzalo, Sitao Huang, Chris Rodrigues and Abdul Dakkak for help in design and coding development of this work. Additionally, I want to thank Carl Pearson, Hee-Seok Kim, Juan Gómez Luna and John Stratton for help in performance portability study and survey.

Also, I want to thank Marie-Pierre Lassiva-Moulin, Andrew Schuh, and Xiaolin Liu for their help, and convey my gratitude to a lot of people I met during my time here, including Wooil Kim, Deming Chen, Nam Sung Kim, Steven Lumetta, and those people I forget to mention or do not know the names of, especially the cleaner of my office.

Finally, I would like to thank my parents and my wife for their unconditional love and support. Thank you everyone.

TABLE OF CONTENTS

LIST OF TABLES vii

LIST OF FIGURES viii

LIST OF ABBREVIATIONS x

CHAPTER 1 INTRODUCTION 1
1.1 Current Practice of Performance Portability 1
1.2 Challenges of Performance Portable Programming 3
1.3 Beyond Performance Portability 4
1.4 Summary of Contributions 5
1.5 Organization of this Dissertation 6

CHAPTER 2 SURVEY OF PERFORMANCE PORTABILITY . . . 7
2.1 Performance Portability Challenges and Conventional Techniques 7
2.2 Current Status of Portable Languages 15

CHAPTER 3 TANGRAM OVERVIEW 21
3.1 Conventional Problem-solving Process 21
3.2 Design Space and Performance Portability 23
3.3 TANGRAM Programming System 26

CHAPTER 4 TANGRAM LANGUAGE 30
4.1 Design Space and Language 30
4.2 TANGRAM Language Design Objectives 32
4.3 Code Example 35
4.4 Recommended Programming Workflow 37
4.5 Comparison of Related Composition-based Languages 37
4.6 Feature Beyond Performance Portability 39

CHAPTER 5 HARDWARE ABSTRACTION AND COMPOSITION 40
5.1 Architectural Hierarchy in Hardware Abstraction 40
5.2 Composition Rule 42
5.3 Composition Specialization 45

5.4 Model Extensibility 49
5.5 Discussion of Hardware Abstraction 50

CHAPTER 6 TANGRAM COMPILER 51
6.1 Compiler, Design Space and Performance Portability 51
6.2 TANGRAM Compiler Design Overview 54
6.3 TANGRAM Compiler Infrastructure 55
6.4 Parser 58
6.5 Analyzer 58
6.6 Composition Planner 63

CHAPTER 7 OPTIMIZATION AND CODE GENERATION 68
7.1 Optimization 69
7.2 Code Generation 71

CHAPTER 8 EVALUATION 75
8.1 Setup 75
8.2 Performance Results 77
8.3 Discussion 83
8.4 Performance Comparison to Existng Composition-based Language 84

CHAPTER 9 CONCLUSION AND FUTURE WORK 86
9.1 Conclusion 86
9.2 Future Work 87

REFERENCES 90

LIST OF TABLES

2.1 Overview of Performance Portability Challenges and Techniques 15
2.2 Overview of Languages and Techniques 20

4.1 Function and Data Qualifiers 34
4.2 Data Manipulation and Parallel Primitives 35

6.1 Relationship between Level Capability to Codelet Property . . 62

7.1 Codegen Preprocessors 72

8.1 Benchmarks 76

LIST OF FIGURES

1.1 OpenCL SGEMM Performance Portability Evaluation 2

2.1 Over-decomposition and Coarsening: Performance comparison of AMD and Intel CPU OpenCL stacks on an i7-3820 CPU. AMD results are used as baselines. 8
2.2 Memory Characteristics and Tiling: Performance of SGEMM with different tiling strategies and tuning parameters on different devices. White and black bars are normalized to different references. Performance relative to reference is compared, not absolute performance. 9
2.3 Atomic Efficiencies and Choices of Algorithms: Stream compaction with shared memory atomics versus Thrust prefix sum. 11
2.4 Reduction with Multiple Levels 12

3.1 Conventional Problem-Solving Abstraction Hierarchy and Corresponding Tasks: At left are levels of abstraction translating a problem into a computational solution. At right are tasks associated with their level of abstraction where they are typically addressed, and a typical boundary between manual and automatic tasks. 22
3.2 Illustration of Relationships among Different Design Points . . 25
3.3 TANGRAM's Problem-Solving Abstraction Hierarchy and Corresponding Tasks: Red, bold-line boxes are different parts from the conventional one. 26
3.4 Big Picture of TANGRAM for Various Devices 27
3.5 TANGRAM's Workflow 28

4.1 Example of Linear Combination 31
4.2 Vectorization for Cooperative and Autonomous Codelets . . . 33
4.3 Codelet Examples for a `sum` Spectrum 36

5.1 Hardware Abstraction of Architectural Levels 40
5.2 Examples and Illustrations of Simple CPU and GPU 41
5.3 Abstract Composition Rules 43

5.4 Example of `map` Rules in OpenMP 44
5.5 Program Composition Rules and Example for Deriving Composition Rules 45
5.6 Rule Specialization and Composition (CPU Example): For the composition plan diagram on the right side, a triangle represents distribution of work according to the partition pattern from the indicated codelet and a circle represents scalar compute according to the indicated codelet. 46
5.7 Rule Specialization and Composition (GPU Example): In addition to the triangles and circles introduced in the previous diagram, a cross represents vector compute according to the indicated codelet. 48
5.8 Example of Design Space for Composition Plans (Showing Four Possible Composition Plans for the Block and Thread Levels) 49

6.1 Performance Impact for Loop Transformations of OpenCL Compilers on an i7-3820 CPU 53
6.2 Performance Impact for Different Vectorization Strategies of Intel CPU OpenCL 54
6.3 TANGRAM Compiler Organization 54
6.4 Composition Algorithm with Pruning 66
6.5 Comparing Composition Rules for Pruning 67

7.1 Heuristic for Data Placement 70
7.2 Codegen for CPU Example in Figure 5.6 73
7.3 Codegen for GPU Example in Figure 5.7 74

8.1 TANGRAM Performance Results 78
8.2 `Scan` Results with or without the Sliding Codelet (Normalized to the Corresponding Thrust Results) 79
8.3 SOP and NOP `SGEMV`-SF Results on GPUs (Normalized to the Corresponding CUBLAS Results) 80
8.4 `SpMV` Results with and without Transposition Optimization (Normalized to the Corresponding References) 82
8.5 `BFS` GPU Performance Breakdown 83
8.6 Comparison between TANGRAM and Petabricks using `DGEMM` on CPU 85

LIST OF ABBREVIATIONS

AMD	Advanced Micro Devices
AST	Abstract Syntax Tree
AVX	Advanced Vector Extension
BFS	Breadth-First Search
BLAS	Basic Linear Algebra Subprograms
C++AMP	C++ Accelerated Massive Parallelism
CEAN	C++ Extension for Array Notation
CPU	Central Processing Unit
CSR	Compressed Sparse Row
CUBLAS	Compute Unified Basic Linear Algebra Subprograms
CUDA	Compute Unified Device Architecture
CUSPARSE	Compute Unified SPARSE Library
DGEMM	Double-precision General Matrix to Matrix Multiplication
FPGA	Field-Programmable Gate Array
GFLOPS	Giga Floating Point Operations Per Second
GPU	Graphics Processing Unit
I/O	Input/Output
ILP	Instruction-Level Parallelism
IR	Intermediate Representation
JDS	Jagged Diagonal Storage

LL/SC	Load-Link/Store-Conditional
MKL	Math Kernel Library
NESL	Nested Data-Parallel Language
NOI	Node of Interest
NOP	Non-Order-Preserving
OpenACC	Open Accelerators
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
SF	Short-and-Fat
SGEMM	Single-precision General Matrix to Matrix multiplication
SGEMV	Single-precision General Matrix-Vector multiplication
SIMD	Single-Instruction Multiple-Data
SM	Streaming Multiprocessor
SOP	Sequential-Order-Preserving
SpMV	Sparse Matrix-Vector multiplication
SSE	Streaming SIMD Extensions
STL	Standard Template Library
TS	Tall-and-Skinny

CHAPTER 1

INTRODUCTION

Modern computing systems are transitioning to heterogeneous platforms, integrating many parallel or specialized computing devices, such as multicore CPUs, GPUs, Intel Xeon Phis, DSPs, and FPGA, to meet computational demands and/or energy efficiency requirements of applications. However, to successfully adopt such computing devices, code transformations and optimizations are typically necessary for gaining performance of existing applications. Ideally, programmers should be able to write in a portable language once, and expect the compilers to automatically perform these code transformations and optimizations and deliver high performance. Yet, the current practice entails substantial effort for rewriting entire applications or core algorithms to gain reasonable performance. Furthermore, maintaining multiple source code versions optimized for different devices could be a burden. Therefore, single-source performance portability has become very desirable for the adoption of heterogeneous systems.

1.1 Current Practice of Performance Portability

Multiple languages, such as OpenCL [1], C++AMP [2], OpenMP [3], and OpenACC [4], have been proposed as a portable parallel language over multiple variants of computing devices. OpenCL, as the most popular proposal, has received tremendous support from both hardware vendors [5, 6, 7, 8, 9, 10, 11, 12, 13] and software developers [14, 15, 16, 17, 18]. However, as a portable language, OpenCL still delivers limited performance compared to vendor libraries [19].

Figure 1.1 shows a quantitative study for performance portability of the current OpenCL stacks by evaluating two versions of Parboil's [20] OpenCL SGEMM benchmark and corresponding vendor libraries on three NVIDIA

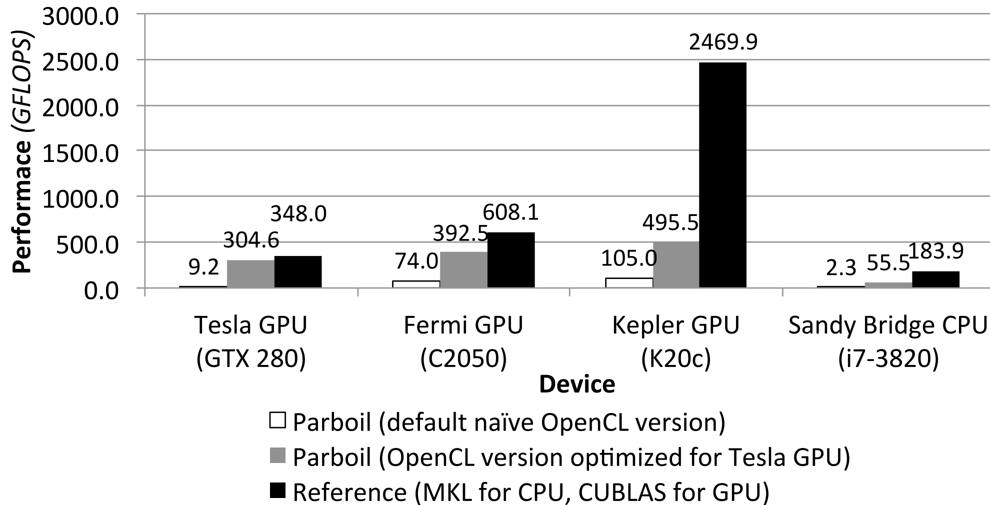


Figure 1.1: OpenCL SGEMM Performance Portability Evaluation

GPUs (Tesla, Fermi and Kepler microarchitectures) and an Intel CPU (Sandy Bridge microarchitecture). Three key observations can be made from these results. First and foremost, though the kernel optimized for a Tesla GPU outperforms the naive version not only for a Tesla GPU but also for Fermi and Kepler GPUs and more interestingly the Sandy Bridge CPU, the kernel optimized for a Tesla GPU only achieves 65% and 20% of the CUBLAS library on the Fermi and Kepler GPUs respectively, and 30% of MKL on the Sandy Bridge CPU. This result demonstrates that optimizations of this OpenCL kernel are not always transferable to non-target GPU and CPU, since, compared to highly optimized vendor libraries, the relative performance drops from 87% to 65%, 20% and 30% respectively. Second, the naive version only delivers limited performance, significantly less than the optimized version (for the Tesla GPU) and the corresponding vendor libraries on the tested GPUs and CPU. This result demonstrates optimizations are necessary for gaining performance in OpenCL. Third, the absolute performance of the kernels does indeed benefit from the architectural improvements from Tesla to Fermi, but the achieved percentage of the vendor library performance decreases. This result demonstrates the improved performance from an architectural upgrade is not sufficient evidence for performance portability. Through this evaluation, we demonstrate the limited performance portability of the current OpenCL stacks.

Qualitatively, OpenCL, as a low-level language, requires optimizations ex-

posed by programmers for gaining performance. Without proper optimizations, an OpenCL kernel delivers very limited performance for most devices. This fact can also be observed by comparing the performance between the white and grey bars on the Tesla GPU in Figure 1.1. The fine-grained thread-level model OpenCL adopts makes performance sensible to target devices, since parallelism becomes the main source of performance scaling. Without carefully crafting work assignment to threads, multi-level tiling, or scheduling of the threads [16, 21, 19], an OpenCL kernel might suffer from huge inefficiency.

On the other hand, optimizations for a specific device are not always transferrable to another device. Also, features specific to the target architecture might not be easily exploited universally. These issues eventually limit performance portability. Ideally, the OpenCL compiler should be able to adapt work assignment to threads, multi-level tiling, or scheduling of the threads for the target device. Unfortunately, techniques for the OpenCL compiler do not yet seem mature enough to deliver reasonable performance portability [19, 22]. Optimizations crafted for the source device are typically coupled together, consequentially obstructing efficient transformation for the target device. A transformed OpenCL kernel typically suffers from huge overhead and inefficiency.

Without architecture-specific customization, performance is limited, while with it, performance portability becomes limited. This dilemma of architecture-neutral programing versus architecture-specific customization becomes the major challenge of performance portability in OpenCL.

1.2 Challenges of Performance Portable Programming

Performance portability is challenging. In the current practice, even with rewriting, programmers still must understand and exploit a wide set of architectural entities to utilize these devices well, not to mention compilers. Most programs come with certain degrees of device-specific optimizations. That makes the programs hard to be analyzed and/or transformed by compilers and eventually obstructs performance portability.

We further classify the challenges of performance portable programming into two major factors, *compiler* and *programming language*. The compiler

factor mainly covers the robustness of optimization in the compiler. For example, most OpenCL compilers [10, 11] focus on loop transformations. The reachability, efficiency and optimality of loop transformations highly impact final performance on a particular device. As mentioned and evaluated in Section 1.1, the OpenCL compilers do not seem robust enough. To the best of our knowledge, even with the most robust OpenCL compiler, there is still a reasonable performance gap between transformed OpenCL code and architecture-specific libraries [19].

On the other hand, the language is mainly about design space. For example, OpenCL forces programmers to express a kernel in a specific, monolithic form. Meanwhile, as a low-level language, it requires most optimizations exposed by programmers for gaining performance. Consequently, the design space of a particular kernel becomes very narrow. In that sense, the optimal design for another device might not be included in the original, narrow space, and that eventually limits performance portability.

In this dissertation, I argue that a new language is required for replacing the current practices of programming systems to achieve practical performance portability. I first demonstrate the limited performance portability of the current practices and argue how conventional optimization techniques get obstructed in these practices. I further identify the main limiting factors of conventional attempts. To overcome this problem, I propose a programming system, called TANGRAM, with a newly-defined high-level language with its corresponding compiler.

1.3 Beyond Performance Portability

Although performance portability is crucial in modern applications, there are several other important factors for a programming system.

Productivity is a measure of the ease to program an application. Typically, most high-level languages provide abstractions to reduce application complexity to achieve high productivity. Modular programming can deliver high productivity by reusing modules.

Maintainability is a measure of the ease to modify code. It typically includes debuggability and incremental improvement. Programs with golden references or test data can have good debuggability. Code reusing for al-

gorithmic improvement and optimizations can deliver incremental improvement.

These factors are not the major foci of this dissertation. Therefore, in the rest of the chapters, we will only briefly discuss these factors, after we introduce the proposed language.

1.4 Summary of Contributions

The following list summarizes the contributions of this dissertation.

- I present a thorough discussion of the challenges of performance portability and a comprehensive survey for existing techniques that tackle these challenges and existing languages that attempt to deliver performance portability.
- I demonstrate the limited performance portability of the current practices, and identify the limiting factors.
- I present a programming language that supports specification of architecture-neutral computations and composition rules.
- I define a simple hardware abstraction model that can be used to specify different architectures with different hierarchical organizations, and show how this model is used to guide architecture-specific composition rules from the proposed architecture-neutral language.
- I design a generic composition algorithm that can be used to compose architecture-specific kernels based on the abstractions.
- I develop a compiler infrastructure specialized for compositions, and use the infrastructure to implement a holistic kernel synthesis framework that leverages our generic composition algorithm, and couples it with other portability techniques during code generation and optimization, such as data placement and parameterization, to synthesize highly optimized processor-specific kernels.
- I demonstrate that kernels synthesized from the same description achieve 70% performance (in the worst observed case) to multiple times performance compared to vendor hand-tuned data-parallel libraries.

1.5 Organization of this Dissertation

The rest of this dissertation is organized as follows. Chapter 2 discusses performance portability challenges and surveys existing corresponding techniques and existing languages. Chapter 3 identifies two major limiting factors (language and compiler) of the current practices and proposes the TANGRAM programming system to resolve these factors. Chapter 4 discusses the language factor and details the TANGRAM language. This chapter also compares the existing languages similar to the TANGRAM language. Chapter 5 introduces a hardware abstraction model, and computation rules, and discusses the relationship between them. Chapter 6 discusses the compiler factor, presents a compiler infrastructure, and details the implementation of the TANGRAM compiler. Chapter 7 explains the optimizations and code generation stages in the TANGRAM compiler. Chapter 8 evaluates the proposed programming system against vendor hand-tuned data-parallel libraries. Chapter 9 concludes this dissertation and outlines areas for continued development.

CHAPTER 2

SURVEY OF PERFORMANCE PORTABILITY

In this chapter, I first introduce the major challenges of performance portability, and corresponding conventional techniques. Then, I briefly introduce the status of the current portable language.

2.1 Performance Portability Challenges and Conventional Techniques

The architectural differences among devices and mismatched optimizations between programs and devices are major sources blocking performance portability. Five major differences, including granularity of hardware thread parallelism, characteristics of the memory subsystem, hierarchical organization of the hardware, size and abundance of resources, and special instructions, are identified. Also, the corresponding conventional techniques and possible obstructions across these differences are discussed in each section.

2.1.1 Granularity of Parallelism

Different devices have different numbers of parallel workers with varying execution resources. For example, multicore CPUs provide a few coarse-grain heavyweight threads, while manycore GPUs provide many fine-grain lightweight threads. Moreover, each CPU thread executes scalar or short vector operations while each GPU thread executes one lane of a long vector operation.

Difference in granularity can be handled portably through over-decomposition and coarsening. Programmers typically describe all the parallelism in applications and leave it for the compiler to decide what to execute in parallel and what to serialize in a single thread (coarsening). OpenCL is one ex-

ample of a programming interface that enables over-expressing parallelism. OpenCL compilers do varying levels of coarsening depending on the device type. Compilers for most GPUs do minor [23] or no coarsening of work-items. Compilers for CPUs coarsen work-items in each work-group down to a single thread [14, 16, 24, 25, 26, 27]. Higher-level languages [28, 29] also support over-decomposition through mapped functions and perform varying degrees of coarsening depending on the target architecture.

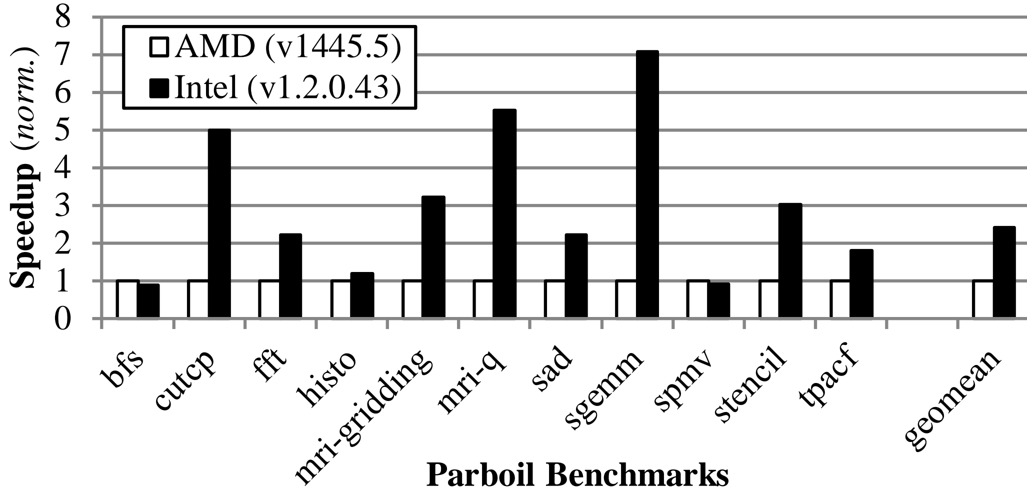


Figure 2.1: Over-decomposition and Coarsening: Performance comparison of AMD and Intel CPU OpenCL stacks on an i7-3820 CPU. AMD results are used as baselines.

There are several design choices of coarsening that might impact its efficiency and overhead. Figure 2.1 compares the performance of AMD's [30] and Intel's [26] CPU OpenCL stacks on an i7-3820 CPU. It is evident that Intel's stack outperforms AMD's. AMD's implementation executes each work-item as a user-level thread, while Intel's coarsens work-items into a single thread. Intel's coarsening enables many optimizations such as collapsing a uniform variable into a single register and vectorizing across multiple work-items. It also reduces thread management overhead and provides more instructions in the same scope for instruction scheduling.

In general, coarsening over-decomposed work can be challenging, because it is not always easy to remove expanded scalars and redundant computations. However, generally automatically coarsening parallel work is still simpler than parallelizing work that is expressed serially [22].

2.1.2 Memory Characteristics

Different types of devices have different memory structures and favor different kinds of locality. For example, CPUs have global memory with private and shared caching, while GPUs - in addition to cached global memory - have scratchpad, read-only, constant, and texture memory. Moreover, CPUs favor intra-thread locality for cache line reuse, whereas GPUs favor inter-thread locality for memory coalescing. Also, in general, CPUs have better temporal locality than GPUs, since the nature of massive multithreading limits lifetime of cache lines on GPUs [31, 32]. Differences in the memory hierarchy also exist across generations of the same device. For example, L1 caches on NVIDIA GPUs were non-existent in the Tesla generation, added in the Fermi generation for caching global memory, and restricted to local memory (register spills, stack data) in the Kepler generation [33].[1]

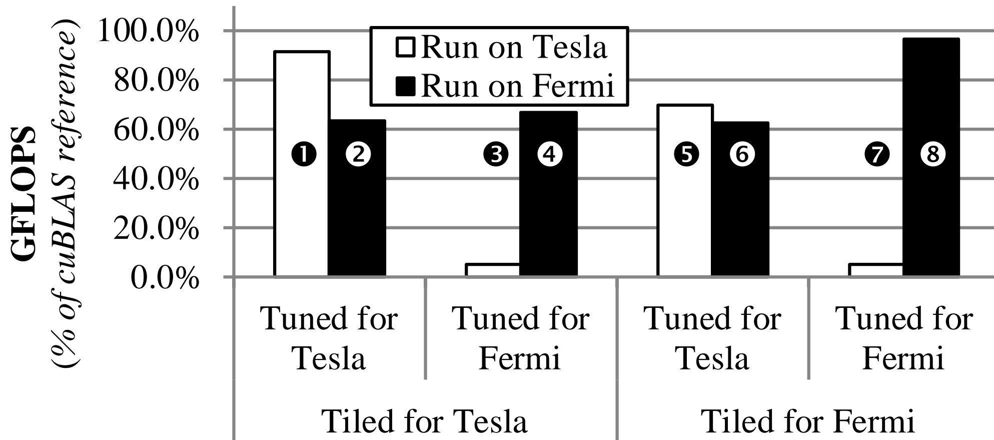


Figure 2.2: Memory Characteristics and Tiling: Performance of SGEMM with different tiling strategies and tuning parameters on different devices. White and black bars are normalized to different references. Performance relative to reference is compared, not absolute performance.

These memory characteristics could highly impact performance of classical optimizations like tiling or other loop transformations. Figure 2.2 shows the performance of SGEMM on different devices using different tiling strategies (e.g., data placement, tile dimensionality, traversal ordering of multidimensional tiles) and tuning parameters (e.g., tile size, work-group size,

[1] Some advanced Kepler GPUs allow L1 caching of global memory via a compiler flag, but it is disabled by default.

coarsening factor). The version tiled for Tesla [34] places the first matrix in registers and the second in shared memory, while the version tiled for Fermi [35] places both in shared memory. They also employ different coarsening strategies. While the version tiled and tuned for Tesla and run on Fermi (②) improves when retuned for Fermi (④ > ②), it improves much more when the data placement and tiling strategy are also changed (⑧ > ④). The opposite is also true (❶ > ❺ > ❼). This result demonstrates the need for proper tiling strategies and data placement (not just retuning) to achieve good performance. Autotuning is discussed separately in Section 2.1.4..

Many techniques have been developed to achieve portability across different memory subsystems. Different types of memories can be utilized with automatic data placement tools such as PORPLE [36], or with heuristic policies [37]. Satisfying different locality preferences can be done by locality-aware scheduling of parallel tasks [16]. Most of these techniques require sophisticated analyses of memory access patterns which could be made easier via built-in data containers [2]. Also, not every memory access is analyzable.

Another example of changes in the memory subsystem is latency of atomics. Scratchpad atomics in Fermi and Kepler GPUs use load-link/store-conditional (LL/SC) instructions, while Maxwell provides faster native instructions for integer scratchpad atomics [38].[2]

These different efficiencies of atomic operations might impact choices of algorithms. Figure 2.3 compares the throughput of stream compaction [39] for two algorithms (with and without atomics) and three different GPU generations. Maxwell's use of native instructions instead of LL/SC significantly improves the algorithm with atomics, causing it to overtake its non-atomics counterpart. Profiling results show that the atomic-based Maxwell version executes significantly fewer instructions than that for Fermi and Kepler. That is because it does not replay instructions until success like LL/SC does. This result further demonstrates the importance of the memory subsystem properties in performance portability. In this case, the difference in atomic latency called for a fundamental change in the optimal algorithm, not just in the placement of data. Algorithm selection is discussed separately in Section 2.1.6.

[2]For floating-point scratchpad atomic operations, Maxwell uses ATOMS.CAS.

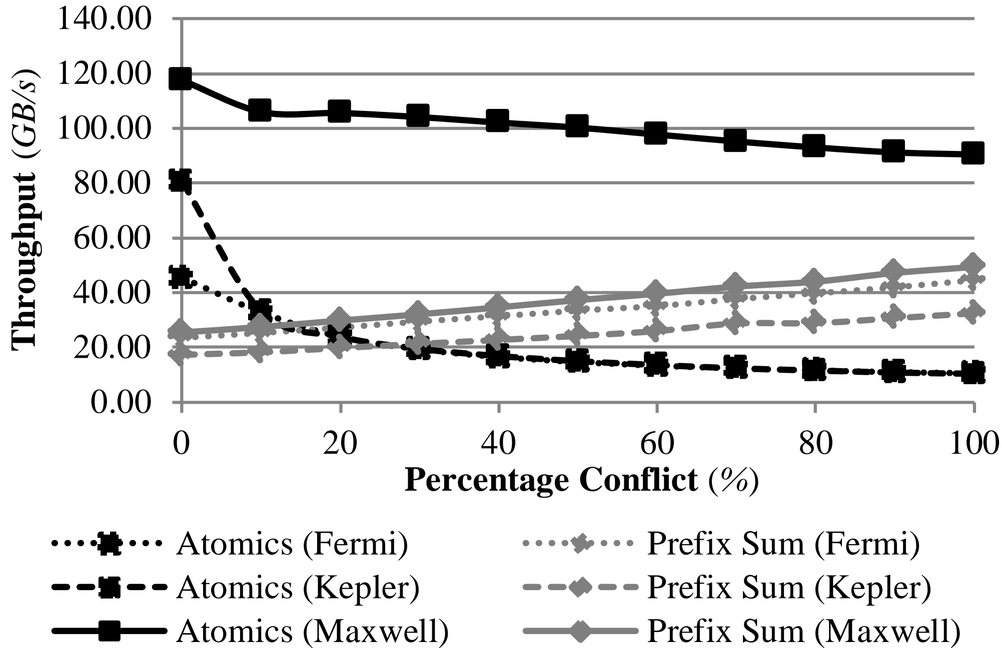


Figure 2.3: Atomic Efficiencies and Choices of Algorithms: Stream compaction with shared memory atomics versus Thrust prefix sum.

2.1.3 Levels of Hierarchy

Different architectures are laid out with varying levels of hierarchy. A multicore CPU has two levels: the sequential thread and the thread pool where all threads are logically equidistant to each other. A GPU has three levels: the work-items, the work-groups, and the work-group range. Work-items in the same work-group are scheduled closer than work-items in different work-groups, and can share local memory and perform lightweight synchronization.

Programming a hierarchical system requires defining multiple phases of computation for each level of the hierarchy. It might impact algorithmic structures significantly. For example, a CPU reduction typically involves a sequential reduction on each thread followed by a collective operation across threads. On the other hand, a GPU reduction involves a sequential reduction in each work-item from multiple work-groups, a tree-reduction across work-items in the same work-group, and another sequential or tree reduction on the partial sums of each work-group.

Figure 2.4 compares the performance of two- and three-level reduction implementations on CPU and GPU. The addition of a third level on CPU

has little impact on performance because a CPU only has two levels. Thus, adding a third level breaks a sequential loop into two nested sequential loops which has a negligible impact on this benchmark. On the other hand, the GPU benefits significantly from the third level because it is a three-level device. The best two-level version does a tree reduction in the work-groups followed by a reduction of partial sums. The third level adds a sequential reduction in each work-item which improves resource utilization. This result demonstrates the importance of considering the hierarchical division of the device for performance portability.

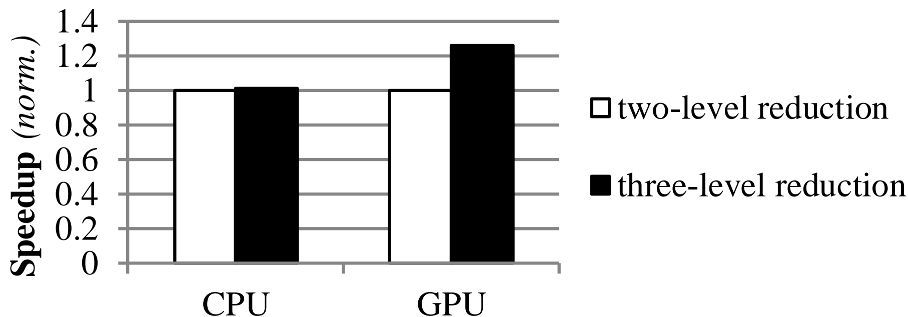


Figure 2.4: Reduction with Multiple Levels

Adapting a program with more levels of hierarchy to an architecture with fewer levels can be done through collapsing levels and coarsening/serialization. However, the reverse action is extremely challenging. For portability, a programming system should remove from the programmer the burden of managing the hierarchy. An effective solution is to allow the programmer to express all possible nested parallelism [40] through recursive and non-recursive implementations of an algorithm. The system then composes these implementations in different ways depending on the underlying architectural hierarchy. A possible challenge of this technique is that it requires sophisticated analyses, transformations, and more work for programmers in many cases.

Dynamic parallelism [41, 42] is one proposal to achieve nested parallelism for GPUs by relying on programmers expressing nested parallelism through certain interfaces and efficiently computing those parallelism through hardware techniques [43, 44, 45] or compiler transformations [46, 47, 48]. Depending on techniques, performance might vary significantly. Also, the current dynamic parallelism only allows a certain type of nested parallelism, instead of general, native nested parallelism.

2.1.4 Resource Constraints

As technology scales, generations of devices come out with different (often increasing) hardware resource sizes and quantities. The size of a resource impacts the tuning of a program in many ways. For example, the optimal loop-tiling factor is usually dependent on the size of a cache, so a larger cache necessitates re-picking the loop tile size. In a GPU, the maximum occupancy of work-items and work-groups on compute units changes across devices, which impacts the optimal work-group size.

Referring back to the results in Figure 2.2, it is evident that within the same tiling strategy, the tuning parameters have a great impact on performance. For both tiling strategies, different devices have different optimal tuning parameters, thus no size fits all. The Tesla-tuned kernels underutilize the maximum occupancy on Fermi (② < ④, ⑥ < ⑧). On the other hand, the Fermi-tuned kernels perform very badly on Tesla because they demand more resources than actually exist which results in register spilling (❸ < ❶, ❼ < ❺).

A common way to deal with variation of resource size is through parameterizing variables that are impacted by the resource size and autotuning those variables at compile time or runtime. Plenty of work has been done on autotuning and search space pruning [49, 50, 51]. However, for a general program, it could be challenging to identify possible variables as parameters, which could be made easier via annotation.

2.1.5 Special Instructions

Some devices provide special instructions to accelerate common computation patterns. These differ across device types and generations. The aforementioned native scratchpad atomics introduced in Maxwell GPUs is an example of such instructions. Other examples include AVX instructions introduced in Sandy Bridge CPUs and shuffle instructions introduced in Kepler GPUs.

One way to handle special instructions is for the compiler to automatically detect common computation patterns and replace them. Some compilers have the ability to detect and replace basic patterns like memory copies, but this approach is not generally applicable due to provability issues. Another approach to handle special instructions is by hiding them behind functions

or language constructs that represent common computation patterns. For example, CEAN [52] is a language extension that enables higher-level expression of vector operations and relies on the compiler to generate SSE, AVX, or serial code depending on what is available, feasible, and efficient. Finally, some instructions may enable completely different algorithms, which requires more general approaches discussed in the next subsection.

2.1.6 General Approaches

Each technique listed so far handles a specific category of architectural differences. A more general approach to performance portability is using libraries of basic data structures and algorithms with algorithm and implementation selection. In this approach, the programmer writes the program using basic computational patterns as building blocks. Programming systems [53, 28, 29, 54, 55] then select the implementation of the pattern that best matches the target device. The alternate implementations can be manually or automatically generated and can differ by implementing different algorithms or different combinations of the techniques in this section applied to the same algorithm.

Algorithm selection can vary in power and flexibility. Computation patterns can be built into the compiler, provided by a library, or defined by the user via a special interface. Selecting between algorithms can be done based on static analysis, static profiling, or dynamic profiling. When source code for the implementation is available, it can be inlined and optimized by the compiler.

2.1.7 Summary of Challenges and Techniques

A summary of the concepts in this section can be found in Table 2.1. In the next section, many of these challenges and techniques are discussed in multiple portable languages.

Table 2.1: Overview of Performance Portability Challenges and Techniques

<table>
<tr><th>Differences</th><th>Specific Techniques</th><th>General Techniques</th></tr>
<tr><td>Granularity of Parallelism</td><td>Over-decomposition
and coarsening</td><td rowspan="5">Basic algorithm libraries
Algorithm selection</td></tr>
<tr><td>Memory Characteristics</td><td>Auto data placement
Locality-aware scheduling</td></tr>
<tr><td>Levels of Hierarchy</td><td>Nested parallelism</td></tr>
<tr><td>Resource Constraints</td><td>Autotuning</td></tr>
<tr><td>Special Instructions</td><td>Language abstraction
Pattern replacement</td></tr>
</table>

2.2 Current Status of Portable Languages

Multiple languages have been proposed as a portable parallel language over multiple variants of computing devices. Here, only mainstream and general-purpose languages are discussed in detail, while some noteworthy domain-specific, mobile-platform, or in-progress languages are briefly mentioned in Section 2.2.5.

2.2.1 OpenCL

OpenCL is the most popular portable language. It is a low-level language, and requires most optimizations to be expressed in kernels by programmers for achieving performance on most devices. Although this design might limit performance, there are still several existing techniques incorporated in OpenCL, benefitting performance portability.

OpenCL adopts a fine-grained thread-level programming model, which natively provides the functionality of over-decomposition. Most OpenCL compilers provide some capabilities of coarsening for CPUs, and few or none for GPUs. In terms of techniques for memory, some recent studies [16, 19, 36, 37] incorporated data placement or locality-aware scheduling into OpenCL, while most vendor compilers do not include these techniques. In terms of hierarchy, OpenCL natively provides a fixed three-level hierarchy, which perfectly fits the current GPUs and applies extra coarsening/serialization to adapt CPUs. For programs with more nested parallelism, dynamic parallelism can be applied for GPUs with hardware techniques [43, 44, 45] or compiler transformations [46, 47, 48] to deliver reasonable performance. However, for architectures with more levels of hierarchy, OpenCL might still have limited

performance. In terms of techniques for resources, most OpenCL compilers provide no such capability. Recent studies [56, 57] proposed such resource management techniques (in CUDA) for GPUs but required either source code or hardware modification. In terms of special instructions, vendor OpenCL compilers might provide some capabilities for their own SIMD instructions. For common primitives, SyCL [58] introduces group-class primitives to bridge performance gaps across OpenCL devices, but it is still in progress. In algorithm selection, OpenCL adopts monolithic expression for kernels, which might limit such support.

As mentioned, OpenCL, as the most popular portable language, has received tremendous hardware and software support. However, as shown in Figure 1.1, the performance portability OpenCL can deliver is still limited. The key reason is that the fixed three-level hierarchy in OpenCL is too GPU-oriented. That might drive entire codebase design biased to GPUs, and eventually limits performance portability to other devices.

2.2.2 C++AMP

C++AMP is a language designed to support data-parallel computation in accelerators, like GPUs, in C++. It relies on concurrency APIs, such as `parallel_for_each`, to label data-parallel code regions, and another keyword `restrict` to specify a region (or a function) to the device where it executes. Different from OpenCL, C++AMP provides a monolithic interface unifying host and device code. Also, C++AMP provides an implicit interface for memory space.

For a data-parallel region, C++AMP adopts fine-grained thread-level programming model through lambda expressions. Therefore, it natively provides the functionality of over-decomposition. Similar to OpenCL, its compilers typically provide some capabilities of coarsening for CPUs, and few or none for GPUs. In terms of techniques for memory, C++AMP relies built-in containers, such as `array_view` and `index`, and tiling APIs to provide hints for data placement. One exception is texture, which requires an explicit keyword. In terms of hierarchy, C++AMP relies on `parallel_for_each`, which natively implies a two-level hierarchy. The keyword `parallel_for_each` implies independent tasks, which potentially can be scheduled freely in a multi-

level hierarchy. Meanwhile, the built-in tiling APIs can further extend its support to three-level devices, such as GPUs. However, since the performance of C++AMP programs seems sensitive to tiling for GPUs, that implies code written for GPUs requires serialization/coarsening for CPUs. It also implies possible performance issues for architectures with more levels of hierarchy. C++AMP compilers do not have its own techniques for resource management. For special instructions, C++AMP might have a limited support, due to no direct support from hardware vendors. Some C++AMP compilers with source-to-source translation could take advantage of backend compilers. Finally, so far, there is no direct support in common primitives or algorithm selection.

In general, C++AMP provides higher-level language features than OpenCL, and can potentially deliver better performance portability. Similar to OpenCL, its main limiting factor is still the support for hierarchy. Different from OpenCL, codebase of C++AMP might not be entirely biased to GPUs. However, code written by programmers still requires explicit optimizations (mainly through tiling) for gaining performance of either CPUs or GPUs. This coding behavior eventually limits performance portability.

2.2.3 OpenMP

OpenMP is a language originally designed for CPU multithreading, and it gets extended for GPUs and other architectures. OpenMP relies on pragmas to annotate code regions for parallelism. Here data parallelism is mainly discussed.

For a data-parallel region, a fine-grained thread-level model is also used. Coarsening can be further controlled through `schedule` annotation. In terms of techniques for memory, OpenMP relies on sophisticated analyses with help of data-sharing attribute clauses, such as `shared`. In terms of hierarchy, OpenMP natively supports a two-level of hierarchy, threads and a range of threads. With the newly introduced `teams` construct, one extra level can be added for GPUs. In terms of resources, OpenMP provides some degree of flexibility for the number of threads (and the size of teams); the runtime or compiler can determine these sizes. For special instructions, OpenMP might have some degree of supports from hardware vendors. Finally, so far, there

is no direct support in common primitives or algorithm selection.

Commonly, the performance of an OpenMP program is very sensitive to its annotation. In order to get good performance, different sets of annotations are required for different devices, or a single set of annotations with different conditions. In the end, no true performance portability is achieved in OpenMP.

2.2.4 OpenACC

OpenACC is a language designed for data-parallel computation in accelerators using annotation like OpenMP. It also uses pragmas to annotate code regions for data parallelism. In terms of discussion related to performance portability, OpenACC is very similar to OpenMP. Compared to OpenMP, OpenACC provides more detailed annotation for loops and data placement. Meanwhile, OpenACC typically has better support in (NVIDIA) GPUs. OpenACC also has limitations similar to those of OpenMP, though it provides better performance.

2.2.5 Other Languages

Here, multiple noteworthy domain-specific, mobile-platform, or in-progress languages are briefly covered. Only main techniques and possible limitation are discussed for each language. Composition-based languages, such as NESL [40], Sequoia [59] and Petabricks [55], are discussed in Chapter 4.

Halide [60] is a domain-specific language in C++ for digital image processing. It adopts most existing techniques discussed in Section 2.1, such as coarsening, auto data placement, locality-aware scheduling, autotuning, and pattern replacement. Particularly for levels of hierarchy, most image processing algorithms have huge independent tasks, which can be arbitrarily scheduled and fitted in a multi-level hierarchy. Though Halide does not provide any functionality of common algorithm library or algorithm selection, its huge number of scheduling choices already provide huge design space for performance portability. Finally, Halide heavily relies on autotuning for selecting the best design.

Surge [28, 29] is a language in C++ for tunable bested data parallelism. It

strongly relies on collective primitives. Each primitive has multiple *schedules* for different levels of hierarchy and tunable parameters. Although general algorithm selection is not applicable in Surge, each built-in primitive could support algorithm selection in its own, Surge applies autotuning (through a machine learning strategy) for selecting the best design.

Parallel Standard Template Library (Parallel STL) [61] is an in-progress built-in library in C++17. It provides a set of parallel templates, which are potentially performance portable across devices. Similar to Surge, each template could have multiple implementations for different levels of hierarchy and tunable parameters. Potentially, compilers could fuse these templates properly and autotuners can select the best design.

RenderScript [62] is a mobile-platform language for Android systems. Compared to OpenCL, which might support some mobile devices, RenderScript typically provides better performance for mobile platform. Although RenderScript provides single-source portability with better performance than OpenCL, RenderScript does not support device selection for programmers. It is clear RenderScript can deliver reasonable degrees of performance portability, but it is difficult to measure its performance portability.

RenderScript does not provide details of its optimization techniques. Since its script (similar to a kernel in OpenCL) is expressed in a fine-grained thread-level model, coarsening most likely is applied. In order to utilize mobile GPUs efficiently, some techniques for memory management could exist.

2.2.6 Summary of Languages and Techniques

A summary of the major languages in this section can be found in Table 2.2. In the next few chapters, I discuss how TANGRAM integrates many of these techniques into a single framework, enables them with language extensions, and manages interactions among them.

Table 2.2: Overview of Languages and Techniques

Types of Techniques	OpenCL	C++AMP	OpenMP	OpenACC
Granularity of Parallelism	✓	✓	✓	✓
Memory Characteristics	△	✓	✓	✓
Levels of Hierarchy	†			
Resource Constraints	*		**	**
Special Instructions	◇	◇	◇	◇
Basic algorithm libraries	[1]		[2]	[2]
Algorithm selection				

✓Support. △Through third-party techniques.

†Through coarsening and dynamic parallelism with third-party techniques.

*Only for work-item sizes. **Only for threads.

◇It depends on the compiler. [1]Through SyCL. [2]Only for reduction pragma.

CHAPTER 3

TANGRAM OVERVIEW

In the previous chapter, multiple challenges, existing techniques, and status of current portable languages were discussed. In this chapter, impact of the above is discussed, in terms of programmers' problems-solving processes. Further, that process leads to the design of the TANGRAM programming system, and how TANGRAM integrates many existing techniques into a single framework, enabling them with language extensions, and managing interactions among them.

3.1 Conventional Problem-solving Process

A typical process for solving a real-world problem on a computing system requires multiple steps. Figure 3.1 summarizes such steps and translates them into levels of abstraction and corresponding tasks for solving real-world problems on computing systems [22]. A problem typically can be translated to a set of reasonable algorithms by domain specialists. Given a computer system, the problem can be further mapped into one or a few specific algorithms through particular algorithmic consideration, such as their algorithmic complexity or corresponding efficiencies on the target system. Then, the algorithm can be translated into a piece of software containing one or multiple computing functions or kernels. Each of these functions or kernels in software could involve multiple optimizations, which could be performed manually by programmers or automatically by a compiler. A boundary between manual and automatic tasks typically happens between the software and compilation levels. Depending on the programming language and the compiler, manual or automatic optimizations could vary. After compilation, the software can execute in a particular computer system.

In general, the computer systems might impact the compiler optimizations,

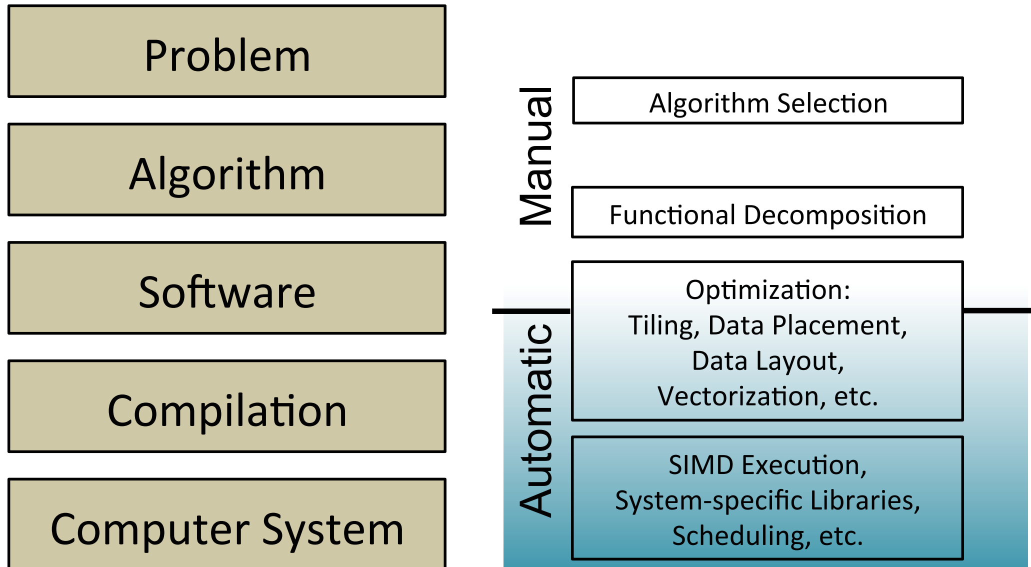


Figure 3.1: Conventional Problem-Solving Abstraction Hierarchy and Corresponding Tasks: At left are levels of abstraction translating a problem into a computational solution. At right are tasks associated with their level of abstraction where they are typically addressed, and a typical boundary between manual and automatic tasks.

the code expression (including manual optimizations) in software, and even the choices of algorithms. The compiler optimizations are highly related to performance portability [22]. Given a precise algorithm, the code expression is also highly related to the language and the compiler, since coding typically requires the interaction between the programmer and the compiler. As discussed in the previous chapter, different portable languages with corresponding compilers might deliver different code expression, leading to various degrees of performance portability. This kind of differences is also a common consideration for programmers' choices of programming languages.

Compared to the former two, the choices of algorithms are typically related to the programmers' understanding of the computer systems, and/or the programmers' consideration for performance portability. For example, given a single-core sequential system, there is no reason for a skilled programmer selecting a parallel algorithm with possibly higher overheads over a sequential algorithm, unless he/she expects this program will be ported to another parallel system. After all, eliminating overheads of a parallel algorithm executing on the single-core sequential system is not trivial.

3.2 Design Space and Performance Portability

The problem-solving process in Figure 3.1 can be considered as a narrowing process of a design space. When algorithms are selected by the programmer, the design space is limited to one particular set. When a particular algorithm is coded in a language, its design space gets narrowed down. When manual optimizations are introduced, the design space shrinks further. Compilation and then execution for a computer system eventually reduce the design space into one specific design point.

Given a problem and corresponding written code in a particular language, a design space S is defined as a set of all possible transformed versions. Since code transformations happen in compilers, and different compilers could deliver different code transformations, the design space becomes a function of a compiler. In order to rule out the effect of compilers in this analysis, we can redefine a design space S as a set of all possible transformed versions *among all possible compilers*. We can also define S_X as the design space for the code written for the computer system X, and S_U as the universal set of design space for all possible machines. Then, we further define $P_{X|S}$ as the best version in the design space S with a particular kind of performance measurement M, assuming the higher is the better, on the computer system X:

$$P_{X|S} = \arg\max_{p \in S} M_X(p),$$

and P_X as the optimal point in the S_U:

$$P_X = P_{X|S_U}.$$

By the definition, P_X can deliver performance better than or equal to $P_{X|S_X}$ on the system X:

$$M_X(P_X) \geq M_X(P_{X|S_X}) \geq M_X(P'_{X|S_X}),$$

where $P'_{X|S_X}$ is defined as the final selected version. Since S_X represents the code written for the system X, $M_X(P_{X|S_X})$ should be reasonably close to $M_X(P_X)$, and $M_X(P'_{X|S_X})$ should be also very close to $M_X(P_{X|S_X})$, if the programmer performs the problem-solving process properly.

When the code with the design space S (regardless it is written for X) is

ported to another system Y, $P_{Y|S}$ can be considered as the best version the system can achieve:

$$M_Y(P_{Y|S}) \geq M_Y(P_{X|S}).$$

Also, $P'_{Y|S}$ can be defined as the selected transformed version:

$$M_Y(P_{Y|S}) \geq M_Y(P'_{Y|S}).$$

Typically, if transformations are profitable, we should follow:

$$M_Y(P'_{Y|S}) \geq M_Y(P'_{X|S}).$$

By the definition, P_Y performs better than $P_{Y|S}$ on Y:

$$M_Y(P_Y) \geq M_Y(P_{Y|S}).$$

Here, we further define (absolute) performance portability (from X) to Y as follows:

$$Perf_Port_Y(S) = M_Y(P'_{Y|S})/M_Y(P_Y) \leq 1.$$

It has an upper bound:

$$Perf_Port_Bound_Y(S) = M_Y(P_{Y|S})/M_Y(P_Y) \leq 1.$$

In general, P_Y might not be in S, so performance portability could not get close to 1.

Note X are not involved in the above two equations. Here, the design space S is determined by the written source code, and our definition of performance portability is only based on the design space S, not the machine X. However, since the source code is written for the machine X, its design space S should be highly biased to the machine X. This bias becomes the major limiting source for performance portability

Depending on the difference between $P'_{Y|S}$ and $P_{Y|S}$ and whether P_Y is in the design space S, performance portability could vary. The former implies the code transformation from the compiler, while the latter comes from the design space, which is highly related to either the written code or even the programming language. It is obvious the code defines the design space. On the other hand, the impact of the language is not that straightforward. Given

a mainstream portable programming languages, such as OpenCL or OpenMP, the programmers might not easily express all of the algorithmic consideration for all various devices. A classic example is that code written in an older version of OpenMP, which only considers two levels of hierarchy for multicore CPUs, cannot perform well on manycore GPUs, without introducing a new `teams` features for three levels of hierarchy. Even with the newest OpenMP or OpenCL, a possible device with four levels of hierarchy could fail to exhibit performance portability.

Figure 3.2 summarizes the relationships among these design points and the design space S. Here, P_Y is the optimal point users desire, while $P'_{Y|S}$ is really the final transformed version. The performance difference between $P'_{Y|S}$ and P_Y reflects performance portability for the system Y. In this illustration, P_Y is not achievable, since it is out of the design space S. Without expanding the design space, the gap between $P'_{Y|S}$ and P_Y cannot be completely removed. On the other hand, $P_{Y|S}$ represents the best version in the design space S. The difference between $P'_{Y|S}$ and $P_{Y|S}$ (and also between $P'_{X|S}$ and $P_{X|S}$) implies the capability of a compiler for performance optimizations. This gap can be removed by improving the compiler.

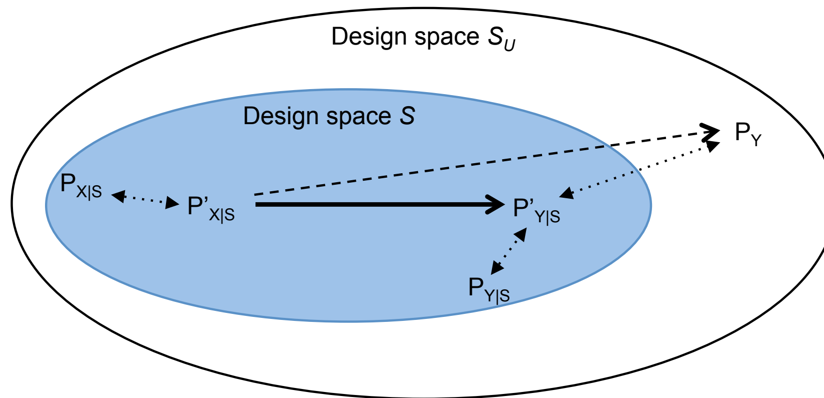


Figure 3.2: Illustration of Relationships among Different Design Points

The **coverage** of a design space is defined as the *cardinality* (or size) of the given design space S over that of the universal design space U:

$$Coverage(S) = |S|/|U|.$$

Here the cardinality of design space is defined as the cardinality of trans-

formed versions from a single codebase in the given programming language. Multiple programming systems [53, 28, 29, 54, 55], as discussed in the previous chapter, provide some degree of support for improving the coverage of an algorithmic design space to bridge possible performance gaps. In order to expand the algorithmic design space, most of them rely on either fixed, implicit functional compositions with built-in libraries, primitives, or specific rules in a domain, or flexible, explicit compositions with new language features. These features inspire me to design a programming system, including defining a new programming language, to natively support effective expression of an algorithmic design space.

3.3 TANGRAM Programming System

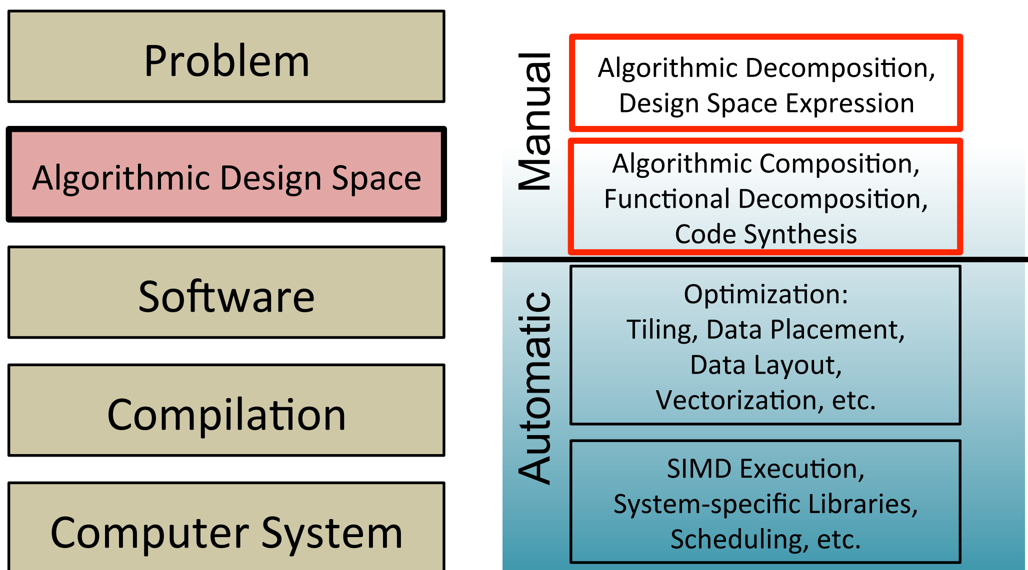


Figure 3.3: TANGRAM's Problem-Solving Abstraction Hierarchy and Corresponding Tasks: Red, bold-line boxes are different parts from the conventional one.

I present TANGRAM, a kernel synthesis framework that generates highly optimized architecture-specific kernels. TANGRAM is designed to achieve performance portability through the following methods:

1. a modular, composition-based programming language that can effectively express a design space with high coverage of implementations

2. a compiler that can effectively explore the design space and facilitate many high-level optimization techniques, described in Section 2.1

In the TANGRAM system, programmers directly express algorithmic design space by writing the proposed high-level language (the TANGRAM language). The language is designed for effective expression of design space through algorithmic decomposition by programmers with the domain knowledge. The TANGRAM compiler then performs design space exploration to select a proper algorithmic structure and implementation, and synthesizes the software with proper high-level optimizations.

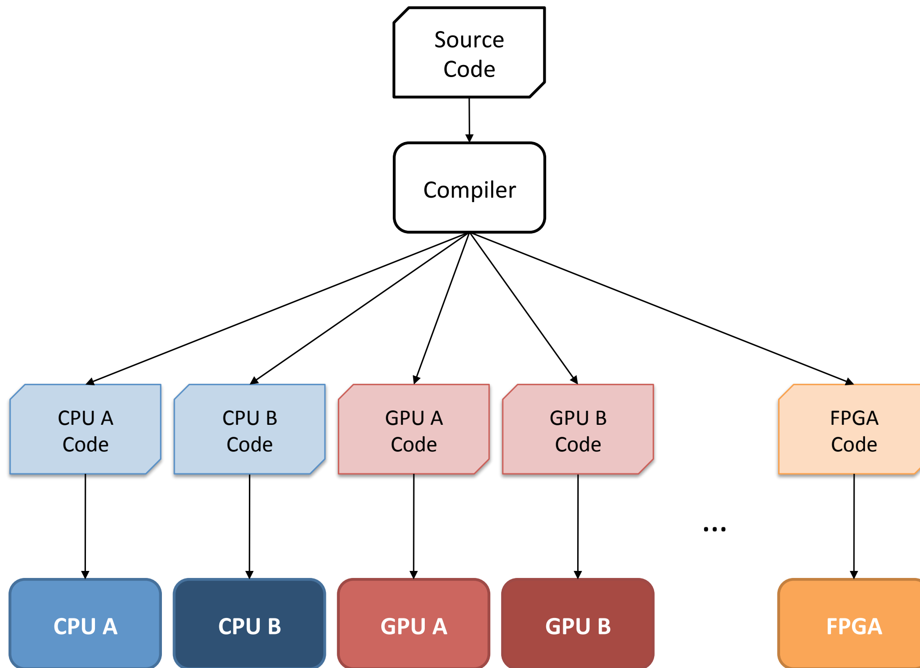


Figure 3.4: Big Picture of TANGRAM for Various Devices

It is not trivial to introduce native support for effective expression of design space. It might change the problem-solving process in Figure 3.1. Figure 3.3 shows the refined version of problem-solving abstraction for TANGRAM. Different from the conventional levels of abstraction, a level of algorithmic design space is introduced to replace conventional level of algorithm. Instead of manually choosing the proper algorithms for a given computer system, programmers express the generic algorithmic design space and rely on the TANGRAM compiler to guide selection for the proper algorithms. Depending on target devices, different output languages/programming models could

be used in the level of software, and corresponding backend compilers in the level of compilation are applied for low-level optimizations.

We argue our refined problem-solving abstraction does not increase complexity for solving a problem compared to the conventional abstraction. In the conventional problem-solving abstraction, programmers are still required to consider algorithmic alternatives in their mind and then make a choice before the software stage. The refined abstraction simply requires programmers to encode their consideration in a certain language.

Figure 3.4 illustrates the big picture of TANGRAM's workflow for various devices. A single copy of input source code written in the TANGRAM language defines the design space for a computation pattern in a given application. Given a device, the TANGRAM compiler further explores the design space and synthesizes one or few kernels. Input source code of the TANGRAM system natively presents a design space, instead of a specific code expression for a machine in conventional programming systems. This strategy potentially can deliver better performance portability.

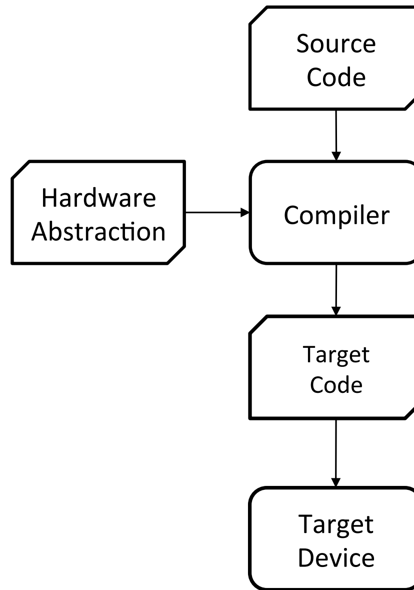


Figure 3.5: TANGRAM's Workflow

Figure 3.5 zooms in and shows the workflow for each device. In the TANGRAM's system, source code is written in TANGRAM language, which will be defined in Chapter 4. Additionally to input source code, *hardware abstraction* is given to the compiler for "understanding" the device, and further

guiding design space exploration. In TANGRAM, hardware abstraction encodes information of architectural hierarchy. It is based on a key observation that the algorithmic structure of a program is highly sensitive to the architectural hierarchy of a device. Each device typically has its own hardware abstraction. The detailed discussion is given in Chapter 5. Most key mechanisms of the TANGRAM system are discussed in the next two chapters. Finally, the TANGRAM compiler is discussed in Chapters 6 and 7. Different from Chapters 4 and 5, which focus on the design and theory, Chapters 6 and 7 cover the implementation details.

3.3.1 Focus of TANGRAM System

The TANGRAM system focuses on language and compiler support for performance portability, and relies on a user-defined design space and hardware abstraction to guide code generation for a highly optimized version. However, in general, given an application and an architecture, the optimal version could depend on runtime information, such as input data distribution, data sizes, resources consumed by co-running applications, or memory which are not accessible statically. Our previous work, DySel [63], is proposed to resolve the above issue by dynamically sampling the performance metrics and then switching versions. Multiple runtime-based approaches [36, 64, 65, 66, 67, 68] also provide the similar functionality to migrate the issue through either performance models or online learning. Although TANGRAM itself does not include runtime, it can work with every existing method, including DySel. Even with a robust runtime that provides lightweight performance monitoring and version switching, a programming system still requires language and compiler support to effectively generate highly optimized versions for switching. TANGRAM is proposed to serve that purpose.

CHAPTER 4

TANGRAM LANGUAGE

In the previous chapters, the relationship between a design space and its performance portability was discussed. The discussion also briefly included the possible impact from the existing language. In this chapter, the detailed relationship between design space and language design is discussed. The design of the TANGRAM language is further defined for effective expression of a design space with high coverage to support performance portability.

4.1 Design Space and Language

In terms of the design space, there are two important factors, **coverage** and **ease of expression**, for a programming language. As mentioned in Section 3.2, the coverage of a design space is defined as the cardinality of transformed versions from a single codebase in the given programming language. Given a language, code might be written in multiple ways, which might deliver different degrees of coverage. In order to define the coverage factor of a language L, the *limit superior* (or upper bound) is applied. That implies $Coverage(L)$ is defined as the *best* $Coverage(S)$ among all possible single-source code written for a particular application.

On the other hand, the **ease of expression** factor typically implies lines of code for expressing a given design space. In mathematics, a space is defined as a set. Here, a design space can be considered as a countable set, which can be either expressed through listing all of its elements or constructed through rules. A systematic construction is highly important for effective expression for a design space. For example, linear combination is used to effectively construct a linear span (which is a vector space) of vectors in another vector

Parts of this chapter appeared in the International Symposium on Microarchitecture [69].The material is used with permission.

space. Figure 4.1 shows an example of linear combination to construct a set containing positive integer pairs (red dots) in 2D Cartesian coordinate system. The set can be expressed by listing all possible points:

$$S = \{(0,0),(2,1),(1,2),(4,2),(3,3),(2,4),...\}.$$

Alternatively, the set can be also expressed by linear combination:

$$S = \{v|v = a*(2,1) + b*(1,2),\ where\, a,b \in N_0\}.$$

Considering this set contains infinite elements, that makes the second method more effective. Similar to this example, in most situations, a design space may contain infinite design elements, making listing impossible, so it requires a set of compositions to effectively express a space.

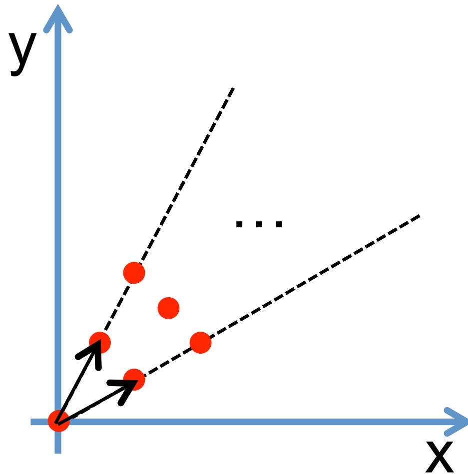


Figure 4.1: Example of Linear Combination

4.1.1 Composition-based Language

Similar to linear combination, composition-based languages [40, 59, 55] or libraries [50, 70] are widely applied to effectively express a program through *modules*. Typically, *modules* are defined through *decomposing* a program to patterns and identifying the reusable ones. Functions or kernels within a program can be composed by combining proper modules.

Multiple composition-based methods enable *functional polymorphism*, which support different implementations or algorithms in the same module. In functional polymorphism, the invoked version can be determined by the programmer, the compiler, the autotuner, or the runtime. Most automatic methods can deliver effective expression with reasonable coverage for the design space of a given single-source codebase.

Compared to listing all possible design elements, composition-based languages generally require an extra step to combine modules. Depending on the method determining the invoked version, it could increase the complexity of compilation, profiling, or runtime. Also, this step could introduce potential overheads, degrading reachable performance.

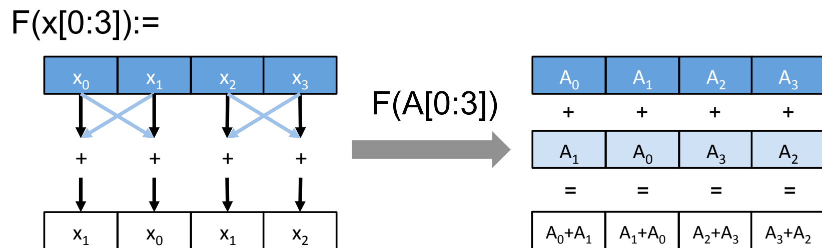
TANGRAM adopts composition-based languages, and applies compiler-based methods to determine invoked modules, and further inlines modules to reduce the invocation overhead. Multiple language features are introduced to improve coverage of design space over existing languages and the capability of the compiler to facilitate both compile-time composition and optimizations.

4.2 TANGRAM Language Design Objectives

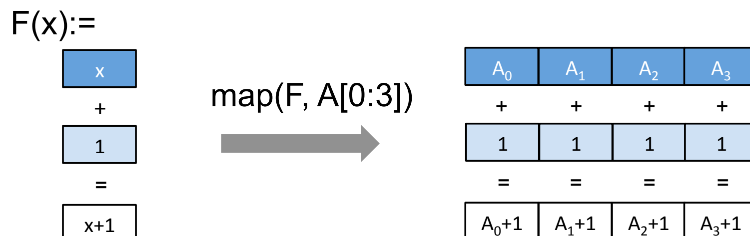
The TANGRAM language is built on top of one key observation that a highly optimized parallel program can be decomposed into multiple simple building blocks, each of which can be frequently applied across data (data parallelism), and/or reusable within the same program and its transformed versions across different devices. The TANGRAM language is designed to support performance portability with the following key objectives:

1. Express equivalent computations interchangeably to expose algorithmic choice and expand design space
2. Express compositions and computations interchangeably to best fit different devices
3. Provide tuning knobs
4. Express data parallelism
5. Ease the analysis of data flow and memory access patterns and the transformation of memory accessing logic

In TANGRAM, the programming model is built around modules called *spectrums* and *codelets*. A *spectrum* represents a unique computation with a defined set of inputs, outputs, and side effects. A *codelet* represents a specific implementation of a spectrum. A spectrum can have many codelets that implement it. These codelets all have the same name and function signature, but can be implemented using different algorithms or the same algorithm with different optimization techniques. With multiple codelets in a spectrum, the design space of this spectrum can be further extended. The interchangeability of the codelets in a spectrum provides functional polymorphism and serves Objective 1.



(a) Cooperative codelet



(b) Autonomous codelet

Figure 4.2: Vectorization for Cooperative and Autonomous Codelets

Codelets are classified into *compound* and *atomic* codelets. *Compound* codelets compose work by invoking primitives and other spectrums (including their own). *Atomic* codelets are self-contained: they compute without further decomposing work or invoking other spectrums. Particularly, *recursive* compound codelets provide capability to variable levels of composition that best fits different devices. The interchangeability of compound and atomic codelets serves Objective 2. Atomic codelets are further classified

into *autonomous* and *cooperative* codelets. Computations in *autonomous* codelets are oblivious to other lanes in the same data parallel computation, whereas computations in *cooperative* codelets can explicitly exchange data with other lanes.

Both autonomous and cooperative codelets can enable vectorization. Figure 4.2 illustrates vectorization in these two kinds of codelets. While autonomous codelets require `map` primitives (discussed in Table 4.2) to enable vectorization, cooperative codelets directly express data exchange within vectorization.

4.2.1 Function and Data Qualifiers

Table 4.1: Function and Data Qualifiers

Qualifier	**Description**
__codelet	Designates that a function declaration is a spectrum, or a function definition is a codelet
__coop	Designates that a codelet is a cooperative codelet
__shared	Designates that a variable is shared across lanes of a cooperative codelet
__tunable	Designates that a variable can be tuned
__tag	Designates the codelet's tag (optional, useful for debugging)
__env	Designates which device(s), if a codelet is specific to a particular device(s)

Table 4.1 summarizes key qualifiers for spectrums, codelets and data in the TANGRAM language. The `__codelet` qualifier is used to designate function declarations as spectrums and function definitions as codelets. The `__coop` qualifier labels cooperative codelets and the `__shared` qualifier labels data structures that are shared by all lanes of a cooperative codelet. The __shared qualifier provides hints for caching and SIMD intrinsics. This `__shared` qualifier is different from the shared memory in CUDA. The data structure can be placed in global memory, shared memory in CUDA, or registers with shuffle instructions in SSE/AVX/CUDA. The `__tunable` qualifier annotates parameters that the compiler can tune serving Objective 3. Through this annotation, the compiler can recognize tunable variables and further enable parameterization based on the granularity of the scheduling, the tiling factor, vector size, and cache line size of the target architecture. The `__tag` qualifier is used to distinguish different codelets with the same function signature in debugging. The `__env` qualifier is optional to enable the user to write device-specific codelets for overcoming the current limitation of the TANGRAM compiler. It can either enable a codelet using an assembly language

for a particular device or disable a certain codelet that is written under an architectural assumption. This qualifier is not intended as the main usage model but is included for completeness. The results we report do not use this feature, but we provide users with the option of using it if they wish (particularly if they want to write device-specific intrinsics or assembly).

4.2.2 Data Manipulation and Parallel Primitives

Table 4.2: Data Manipulation and Parallel Primitives

Primitives	**Description**
map(*f*, *c*)	Returns a container where each element results from applying spectrum *f* to each element in container *c*
partition(*c*, *n*, *start*, *inc*, *end*)	Returns *n* sub-containers c_i of *c* where c_i goes from *start[i]* to *end[i]* with increment *inc[i]*
sequence(*a*)	Returns an integer sequence of the value of *a* (argument to partition)
sequence(*start*, *inc*, *end*)	Returns a sequence of integers from *start* to *end* with increment *inc* (argument to partition)
sequence(*c*, *start*, *inc*, *end*)	Returns a sequence of integers from values of *c* at indexes *start* to *end* with increment *inc* (arg. to partition)
Array<*n*, *type*>	A *n*-dimensional container of values of type *type*

The TANGRAM language comes with several built-in primitives listed in Table 4.2. A `map` primitive is used to express data parallelism by applying a codelet to all elements of a data container, serving Objective 4. This primitive is crucial to enable multithreading and vectorization with autonomous codelets. The `partition` primitive is used to express the pattern used for data partitioning. The three `sequence` primitives are used as arguments to `partition` to express different patterns. These primitives along with the `Array` container are used to facilitate memory access and data flow analysis serving Objective 5.

4.3 Code Example

Figure 4.3 shows an example of four codelets implementing a spectrum for computing a summation. All codelets have the same function signature and are marked with the `__codelet` qualifier. Figure 4.3(a) shows an atomic autonomous codelet where each lane performs the summation sequentially. Figure 4.3(b) shows an atomic cooperative codelet with the `__coop` qualifier that performs a tree-based summation among lanes of parallel execution.

The codelet contains multiple variables and arrays that are shared across the lanes and are marked with the `__shared` qualifier. Special functions `coopIdx()` and `coopDim()` are used to obtain the lane ID and the width of the cooperative codelet respectively.

```
__codelet
int sum(const Array<1,int> in) {
  unsigned len = in.size();
  int accum = 0;
  for(unsigned i=0; i < len; ++i) {
    accum += in[i];
  }
  return accum;
}
```

(a) Atomic autonomous codelet

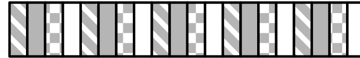
```
__codelet __coop __tag(kog)
int sum(const Array<1,int> in) {
  __shared int tmp[coopDim()];
  unsigned len = in.size();
  unsigned id = coopIdx();
  tmp[id] = (id < len)? in[id] : 0;
  for(unsigned s=1; s<coopDim(); s *= 2) {
    if(id >= s)
      tmp[id] += tmp[id - s];
  }
  return tmp[coopDim()-1];
}
```

(b) Atomic cooperative codelet

```
__codelet  __tag(asso_tiled)
int sum(const Array<1,int> in) {
  __tunable unsigned p;
  unsigned len = in.size();
  unsigned tile = (len+p-1)/p;
  return sum( map( sum, partition(in,
      p,sequence(0,tile,len),sequence(1),sequence(tile,tile,len+1))));
}
```

(c) Compound codelet using adjacent tiling

```
__codelet __tag(stride_tiled)
int sum(const Array<1,int> in) {
  __tunable unsigned p;
  unsigned len = in.size();
  unsigned tile = (len+p-1)/p;
  return sum( map( sum, partition(in,
      p,sequence(0,1,p),sequence(p),sequence((p-1)*tile,1,len+1))));
}
```



(d) Compound codelet using strided tiling

Figure 4.3: Codelet Examples for a `sum` Spectrum

Figures 4.3(c) and 4.3(d) show compound codelets using different tiling strategies. They both contain a tunable variable `p` that controls the number of partitions in the recursive call. In Figure 4.3(c), the array is partitioned into adjacent contiguous tiles that start `tile` elements apart and have an internal stride of one. Such a partitioning is suitable for distributing data to workers with different caches such as different CPU threads or GPU thread blocks. In Figure 4.3(d), the array is partitioned into interleaved tiles that are staggered to start one element apart and have an internal stride of `p`. Such a partitioning is suitable for distributing data to workers that execute

together and in the same cache such as CPU vector lanes or GPU threads.

4.4 Recommended Programming Workflow

As discussed in Figure 3.3, in a typical TANGRAM workflow, programmers identify the main computations in a program and declare a spectrum for each. Programmers are encouraged to write an autonomous codelet for each spectrum then a straightforward decomposition. After that, programmers can revise the spectrum by exploring alternative algorithms and decompositions. Adding new codelets is the main venue for improving the achieved performance on existing and new processors. For example, one could add a new atomic codelet to the `sum` spectrum in Figure 4.3 that uses atomic read-modify-write operations to accumulate the element values into a shared accumulator variable. This codelet may be more efficient than the codelet in Figure 4.3(b) if the processor supports very high-bandwidth execution of atomic read-modify-write operations.

In TANGRAM, each codelet is intended to represent a fundamentally different algorithm or composition. Thus, it is unlikely that there will be more than a handful of atomic codelets in each spectrum. We do not expect the number of codelets to increase significantly over the lifetime of the code base.

4.5 Comparison of Related Composition-based Languages

As discussed in Section 4.1.1, multiple composition-based languages have attempted to deliver both high coverage and easy expression for a design space. In this section, multiple noticeable composition-based languages are compared with the TANGRAM language in terms of *language* features for performance portability. Here, we mainly discuss the language features, unless a compiler feature merits mention. Multiple noticeable related libraries and languages internally with composition capabilities will be also mentioned and compared.

NESL [40] provides explicit composition capability with multiple interchangeable computations, similar to TANGRAM. Its composition is mainly

driven through problem (input and/or output) sizes expressed by programmers, while TANGRAM's composition is commonly with tunable qualifiers and mainly driven through the compiler. In general, NESL has no performance tuning functionality, while TANGRAM natively supports performance tuning through `__tunable` qualifiers and parameterization. NESL provides not only data manipulation/parallel primitives, but also *computation* primitives, such as *scan* and *reduce*, while TANGRAM only provides data manipulation and parallel primitives. In terms of vectorization, NESL relies on auto-vectorization and computation primitives, while TANGRAM provides auto-vectorization plus cooperative codelets for natively expressing sophisticated vector algorithms.

Sequoia [59] relies on compositions mainly for adapting memory hierarchies across architectures. Similar to TANGRAM, in Sequoia, programmers describe inner tasks for composition rules and leaf tasks for leaf computation of the lowest hierarchy. Multiple hierarchies can be adapted by iterating inner tasks recursively. Sequoia also provides tunable keywords for both parameter tuning and compositions. In terms of languages, the major difference between Sequoia and TANGRAM is the cooperative codelet for natively expressing vector algorithms. Without native support for vector algorithms, Sequoia might not express a design space well in some applications.

Petabricks [55] applies compositions for algorithmic design space search. Given a problem, different problem (input and/or output) sizes might prefer different algorithms. Compositions can simplify design space search through dynamic programming and also enable parallelism. Although Petabricks and Sequoia have different design purposes, they share very a similar language structure. Compared to TANGRAM, Petabricks also lacks of functionality to express vector algorithms. It is also worth mentioning that Petabricks only integrates few architectural optimizations in its compiler, and mainly focuses on autotuning for the best algorithm.

Multiple libraries, such as **Spiral** [50] or **GotoBLAS** [70], apply similar composition strategies directly using domain-specific algebras. All functionalities, such as functional polymorphism, vectorization, parameter tuning, and optimizations can be achieved through mapping domain-specific algebras to low-level source code.

Languages like **Halide** [60] or **Surge** [28, 29] provide implicit composition rules for basic operations, such as loop scheduling or primitives, and

deliver functional polymorphism. Halide introduces various scheduling policies for stencil operations specifically for 2D image processing applications. Surge adopts basic collective primitives for constructing applications. Each collective primitive has its own set of scheduling policies and implementations. Neither Halide nor Surge provides explicit composition capability to programmers, but internally the composition might be applied in scheduling policies or implementations.

4.6 Feature Beyond Performance Portability

As mentioned in Section 1.3, beyond performance portability, productivity and maintainability are also crucial for a programming system.

TANGRAM is a high-level language abstracting computation from architectural details, with a rich set of data and parallel manipulation primitives to reduce coding complexity. Meanwhile, TANGRAM adopts modular programming for computation patterns, which are reusable within a program and the transformed versions across different devices. Therefore, TANGRAM can deliver reasonable productivity.

On the other hand, TANGRAM is designed with user debugging in mind. Since optimizations are decoupled from logic, programmers express high-level intent rather than coding architectural details, which may introduce bugs and limit portability. Since codelets within a spectrum are supposed to be functionally equivalent, interchangeable codelets might be used for verification. As mentioned in Section 4.4, codelets can be built incrementally, and that is considered as an incremental improvement for a program, While other languages may require programmers to rewrite the kernel to exploit a new architectural feature, TANGRAM only requires adding new interchangeable codelets that materialize the feature. In this process, minimal code changes are required to adapt to new architectures. These all features serve maintainability.

CHAPTER 5

HARDWARE ABSTRACTION AND COMPOSITION

Different architectures might prefer different compositions due to different architectural hierarchies, leading to different implementations. A hardware abstraction is proposed to model architectural hierarchy and to further guide a composition plan.

5.1 Architectural Hierarchy in Hardware Abstraction

$L := C_L, (\ell_L, S_L)$
L : level
C : computational capability (possible values: SE – scalar execution, VE – vector execution)
(ℓ, S): (subordinate level , capability to synchronize subordinate level)

Figure 5.1: Hardware Abstraction of Architectural Levels

TANGRAM's approach builds on the observation that a key differentiating factor between devices is their architectural hierarchy. Different devices come with different architectural levels. For example, a simple CPU may be modeled as two-level devices (process, thread), while a simple GPU may be modeled as three-level devices (grid, block, thread). Here, the CPU SIMD unit and the GPU warp are omitted for brevity and clarity, but discussed later in Section 5.4.

Each architectural level may have the ability to execute scalar or vector code. Such a computational capability is represented by C in Figure 5.1. Furthermore, each level of the hierarchy having a level beneath it has the capability to distribute workloads and synchronize across the elements of that level. For example, a process can undergo barrier synchronization among all its threads. The subordinate level and synchronization capability are

Parts of this chapter appeared in the International Symposium on Microarchitecture [69]. The material is used with permission.

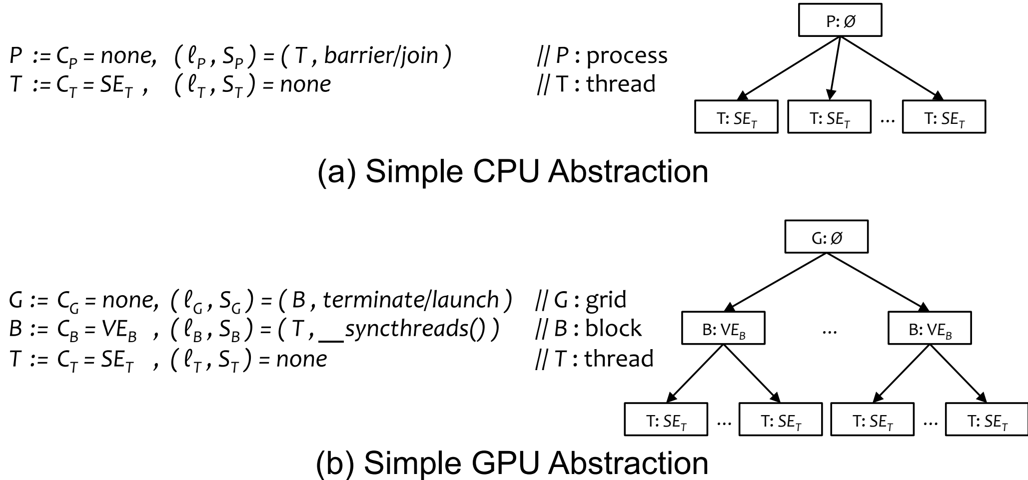


Figure 5.2: Examples and Illustrations of Simple CPU and GPU

denoted by (ℓ, S). Figure 5.2 shows examples of architectural hierarchies in a simple CPU and GPU, and corresponding illustrations. The simple CPU is treated as a two-level device (without SIMD units): the first level being the process P and the second being the thread T. The process does not have a computational capability, but has subordinate threads and the ability to synchronize those threads via a barrier/join operation. The thread has scalar execution capability and has no subordinate levels. Similarly, the simple GPU is treated as a three-level device. The grid G level has no computational capability, but has a subordinate block level B and can perform a barrier synchronization across blocks via kernel termination and launch of a new kernel. The block level has a vector execution capability, a thread T subordinate level, and can synchronize subordinate threads using `__syncthreads()`. Finally, the thread level has scalar execution capability and no subordinate level.

In TANGRAM, hardware abstraction serves two major purposes for composition. First, it defines legal composition rules for a given architecture. Section 5.3 will explain this purpose in detail. Second, it also guides the synthesis of composition plans, which will be discussed in the next chapter.

5.2 Composition Rule

Composition rules are the main cores of TANGRAM. They impact both the design space and the code transformations. In TANGRAM, each codelet typically implies a composition rule. Meanwhile, each spectrum implicitly has a few composition rules, while every primitive also has its own built-in composition rules. Some primitives might have more than one rule.

Before hardware abstraction is specified, composition rules are architecture-neutral. These architecture-neutral composition rules can be classified into two categories. They are *abstract composition rules* (Section 5.2.1) pre-defined in the TANGRAM programming model and *program composition rules* (Section 5.2.2) expressed by the programmer.

5.2.1 Abstract Composition Rule

Given a level of hierarchy, we can introduce multiple abstract composition rules that are used to extract program composition rules from the codelets. Abstract composition rules can be applied to different objectives, including a spectrum, a codelet or a primitive. Figure 5.3 summarizes all of the abstract composition rules, and shows corresponding notations, which will be used in the following sections.

For a spectrum, two rules can be specified. First, **Select** composes a spectrum at a level by selecting a codelet of that spectrum. Second, **Devolve** composes a spectrum at a level by synchronizing then delegating the spectrum to a single worker of that level's subordinate level. For example, a master thread may perform a task on behalf of all threads in a process. These two rules can be distinguished by the level of hierarchy. Particularly, when the Devolve rule happens for a spectrum at a level, that spectrum will be scheduled to that level's subordinate level, and that spectrum at the subordinate level requires another rule to be specified.

When the Select rule happens in a spectrum at level, a codelet is chosen. If the chosen one is a compound codelet, its body will be traversed and the corresponding invoked spectrums in the body will be composed. On the other hand, if the chosen codelet is an atomic codelet, the following rule might be specified. **Compute** composes an atomic codelet at a level by assigning that atomic codelet to that level's computational capability (if possible). Note

For Spectrums only:

Select: *compose(s , L)*
→ *compose(c , L)* // *s* : spectrum, *c* : codelet of spectrum *s*

Devolve: *compose(s, L)*
→ S_L , *devolve*(ℓ_L) , *compose*(s, ℓ_L) // *s* : spectrum

For Codelets only:

Compute: *compose(c, L)*
→ *compute*(c , C_L) // *c* : atomic codelet

For Partition Primitives only:

Regroup: *compose(partition(…, p), L)*
→ S_L , *regroup(p, L)* // *p* : a partitioning scheme

For Map Primitives only:

Distribute: *compose(map(s, …), L)*
→ *distribute*(ℓ_L) , *compose*(s, ℓ_L) // *s* : spectrum

Serialize: *compose(map(s, …), L)*
→ *serial(L) , compose(s, L)* // *s* : spectrum

Split: *compose(map(s, …), L)*
→ *split(…,L), compose(map(map(s), …), L)* // *s* : spectrum

For Spectrums and Primitives:

Cascade: *compose(f(g(…), L)*
→ *compose(g(…), L) , compose(f(…), L)* // *f, g* : primitives or spectrum invocations

Figure 5.3: Abstract Composition Rules

this rule can be specified only when the chosen atomic codelet meets the level's computational capability.

For a primitive, a composition rule is determined by the primitive's type. There are two major sets, one for `partition` primitives and one for `map` primitives. For a `partition` primitive, **Regroup** composes the primitive at a level by synchronizing to ensure the data is ready, then regrouping the data at that level according to the partitioning scheme.

For a `map` primitive, three rules can be specified. First, **Distribute** composes the primitive at a level by spawning multiple workers of the subordinate level and distributing the spectrum to those workers. Second, **Serialize** composes the primitive at a level by creating a loop at the level (if it has a computational capability) to serialize the `map` operation. Third, **Split** composes the primitive at a level by breaking the `map` into a composition of two `map`s. Both inner and outer `map`s require new rules to be specified. This strategy is typically applied to extend the reach of the original `map` to a lower subordinate level by specifying the **Distribute** rule to the outer `map`. Figure 5.4 shows an example of these rules in OpenMP. These rules of `map`

can be considered as different loop transformations for a parallelizable loop. Therefore, we also call these rules *scheduling policies*. For brevity and clarity, the Serialize and Split rules are omitted in examples of this chapter, but are discussed in later chapters.

```
B = map(F, A[0:15]);
```

(a) Input code

```
#pragma omp parallel for
For(i = 0:15) {
    B[i] = F(A[i]);
}
```

(b) Transformed code for **Distribute**

```
For(i = 0:15) {
    B[i] = F(A[i]);
}
```

(c) Transformed code for **Serialize**

```
//undetermined
For(i = 0:3) {
    //undetermined
    for(j = 0:3) {
        B[ i * 4 + j ] = F(A[ i * 4 + j ]);
    }
}
```

(d) Transformed code for **Split**

Figure 5.4: Example of `map` Rules in OpenMP

Last, when cascaded primitive or spectrum invocations are composed at a level, **Cascade** composes them sequentially at that level. The order of compositions is the same as the order of invocations, from inner to outer ones.

5.2.2 Program Composition Rule

The composition rules of a program are extracted by applying the abstract rules to the codelets until none can be applied deterministically anymore. Typically, one program composition rule is extracted per codelet. Figure 5.5 shows the rules extracted from the codelets in Figure 4.3. Rule 1 is basically the Devolve rule. Rules 2 and 3 show how atomic codelets generate rules that assign those codelets to computational capabilities (autonomous to scalar, cooperative to vector). Rules 4 and 5 show how compound codelets generate more complex rules corresponding to their functionality. Rule 1 is extracted from the devolve abstract rule which requires no codelets.

Program Composition Rules: (for the sum example)

Rule 1: *compose(sum , L)* // Derived from Devolve
→ *S_L , devolve(ℓ_L) , compose(sum, ℓ_L)*

Rule 2: *compose(sum , L)* // Derived from codelet a (c_a)
→ *compute(c_a , SE_L)*

Rule 3: *compose(sum , L)* // Derived from codelet b (c_b)
→ *compute(c_b , VE_L)*

Rule 4: *compose(sum , L)* // Derived from codelet c (c_c)
→ *S_L , regroup(p_c, L) , distribute(ℓ_L) , compose(sum, ℓ_L) , compose(sum, L)*

Rule 5: *compose(sum , L)* // Derived from codelet d (c_d)
→ *S_L , regroup(p_d, L) , distribute(ℓ_L) , compose(sum, ℓ_L) , compose(sum, L)*

Example for Deriving Composition Rules from Compound Codelets: (using codelet c as an example)

compose(sum , L)
→ *compose(c_c , L)* // Rule: Select
→ *compose(sum(map(sum, partition(…, p_c))), L)* // Expand c_c
→ *compose(map(sum, partition(…, p_c)) , L) , compose(sum, L)* // Rule: Cascade
→ *compose(partition(…, p_c) , L) , compose(map(sum, …) , L) , compose(sum , L)* // Rule: Cascade
→ *S_L , regroup(p_c, L) , compose(map(sum, …), L) , compose(sum, L)* // Rules: Regroup
→ *S_L , regroup(p_c, L) , distribute(ℓ_L) , compose(sum, ℓ_L) , compose(sum, L)* // Rules: Distribute

Figure 5.5: Program Composition Rules and Example for Deriving Composition Rules

An example of how rules are extracted from a compound codelet is also shown in Figure 5.5 for codelet c (Figure 4.3(c)). In the resulting rule, the data is first regrouped according to the scheme specified in codelet c (p_c). The inner `sum` (the argument to the `map` primitive) is then distributed to multiple workers of the subordinate level. Finally, the outer `sum` is performed at the original level. Here, as mentioned, the `map` primitive allows the Distribute only for brevity and clarity. In practice, the derivation should stop at the Regroup rule, and we label the different parts with a light blue color.

Up until this point, the composition rules have been generated from the codelets without any consideration for the target device. This indicates the architecture-neutrality of the programming model.

5.3 Composition Specialization

In this section, we show how the program composition rules (Section 5.2.2) are specialized when architectures are specified through hardware abstraction (Section 5.1). We use the CPU and GPU examples in Figure 5.2 to assist

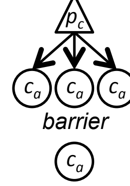
with the explanation.

5.3.1 CPU Example

Hardware Abstraction:

P := C_P = none, (ℓ_P , S_P) = (*T* , barrier/join) // *P* : process

T := $C_T = SE_T$, (ℓ_T , S_T) = none // *T* : thread



Specialized Composition Rules:

P rules: P1: *compose(sum , P)*
→ S_P , *devolve(T) , compose(sum, T)*

P4: *compose(sum , P)*
→ S_P , *regroup*(p_c, *P) , distribute(T) , compose(sum, T) , compose(sum, P)*

P5: *compose(sum , P)*
→ S_P , *regroup*(p_d, *P) , distribute(T) , compose(sum, T) , compose(sum, P)*

T rules: T2: *compose(sum , T)*
→ *compute*(c_a , SE_T)

Composition Example:

compose(sum , P)
→ S_P , *regroup*(p_c, *P) , distribute(T) , compose(sum, T) , compose(sum, P)* // *P4*
→ S_P , *regroup*(p_c, *P) , distribute(T) , compute*(c_a , SE_T) , S_P , *devolve(T) , compose(sum, T)* // *T2 , P1*
→ S_P , *regroup*(p_c, *P) , distribute(T) , compute*(c_a , SE_T) , S_P , *devolve(T) , compute*(c_a , SE_T) // *T2*

Figure 5.6: Rule Specialization and Composition (CPU Example): For the composition plan diagram on the right side, a triangle represents distribution of work according to the partition pattern from the indicated codelet and a circle represents scalar compute according to the indicated codelet.

Figure 5.6 shows how the composition rules can be specialized for the simple CPU specified in Figure 5.2. Based on this architecture abstraction, the extracted rules 1 through 5 in Figure 5.5 can be specialized for the CPU architecture. For the process level P, only rules 1, 4, and 5 can be specialized. Rules 2 and 3 cannot because the process does not have a compute capability; it can only distribute work to its subordinate (thread) level. For the thread level T, only rule 2 can be specialized. Rules 1, 4, and 5 cannot because the thread does not have a subordinate level in the specification, and rule 3 cannot because the thread does not have SIMD units (vector computational capability) needed to execute cooperative codelets. The resulting specialized rules are $P1$, $P4$, and $P5$ for the process level and $T2$ for the thread level in Figure 5.6.

To assist the reader with understanding the application of these rules,

Figure 5.6 also shows an example of one possible composition plan that they can be used to derive. This plan takes three steps to create. First, $P4$ is applied to distribute the `sum` to the different threads and then to sum up the partial sums at the process level. Next, $T2$ is used to perform the `sum` in each thread. Also, because the process cannot perform computations, $P1$ is used to delegate one of its threads to perform the final `sum` of partial sums. Finally, $T2$ is used to perform that delegated `sum` using a single thread. The created composition plan is illustrated with the diagram on the right side of Figure 5.6.

5.3.2 GPU Example

Figure 5.7 shows a similar example for the simple GPU in Figure 5.2. Similar to the CPU example, rules 1, 4, and 5 can only be assigned to levels with subordinate levels, rule 2 to levels with scalar execution capability, and rule 3 to levels with vector execution capability. Accordingly, we can specialize rules 1, 4, and 5 for the grid level G, rules 1, 3, 4, and 5 for the block level B, and rule 2 for the thread level T.

Figure 5.7 also shows an example of one possible composition plan that these specialized rules can be used to create. A brief explanation for this composition is given since the process is similar to that shown in the CPU example. $G4$ is first applied to distribute the `sum` to the block levels and then to sum up the partial sums at the grid level. $B4$ and $G1$ are then applied to the first and second `sum`s and distribute workloads to the thread and block levels respectively. Next, $T2$ and $B3$ are used to perform the first and second delegated `sum`s in each thread and block respectively, while $B4$ is applied to the third `sum` and distributes workloads to the thread level. Finally, $T2$ and $B3$ are used again to perform the first and second delegated `sum`s in each thread and block respectively.

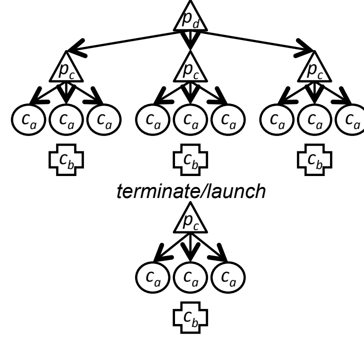
5.3.3 Composition and Design Space

In the previous cases, we only show one possible composition plan. In general, multiple legal composition plans for a given architecture form a design space. Figure 5.8 shows an example of a possible design space for composing the `sum`

Hardware Abstraction:

G := C_G = none, (ℓ_G , S_G) = (B , terminate/launch) // G : grid
B := C_B = VE_B , (ℓ_B , S_B) = (T , __syncthreads()) // B : block
T := C_T = SE_T , (ℓ_T , S_T) = none // T : thread

Specialized Composition Rules:



G rules: G1: *compose(sum , G)*
→ S_G , *devolve(B) , compose(sum, B)*
G4: *compose(sum , G)*
→ S_G , *regroup(*p_c*, G) , distribute(B) ,*
compose(sum, B) , compose(sum, G)
G5: *compose(sum , G)*
→ S_G , *regroup(*p_d*, G) , distribute(B) ,*
compose(sum, B) , compose(sum, G)

B rules: B1: *compose(sum , B)*
→ S_B , *devolve(T) , compose(sum, T)*
B3: *compose(sum , B)*
→ *compute (*c_b , VE_B*)*
B4: *compose(sum , B)*
→ S_B , *regroup(*p_c*, B) , distribute(T) , compose(sum, T) , compose(sum, B)*
B5: *compose(sum , B)*
→ S_B , *regroup(*p_d*, B) , distribute(T) , compose(sum, T) , compose(sum, B)*

T rules: T2: *compose(sum , T)*
→ *compute(*c_a , SE_T*)*

Composition Example:

compose(sum , G)
→ S_G , *regroup(*p_c*, G) , distribute(B) , compose(sum, B) , compose(sum, G)* // G4
→ S_G , *regroup(*p_c*, G) , distribute(B) ,* S_B *, regroup(*p_d*, B) , distribute(T) ,* // B5 , G1
compose(sum, T) , compose(sum, B) , S_G *, devolve(B) , compose(sum, B)*
→ S_G , *regroup(*p_c*, G) , distribute(B) ,* S_B *, regroup(*p_d*, B) , distribute(T) ,* // T2 , B3, B5
compute(c_a , SE_T*) , compute (*c_b , VE_B*) ,* S_G *, devolve(B) ,* S_B *, regroup(*p_d*, B) ,*
distribute(T) , compose(sum, T) , compose(sum, B)
→ S_G , *regroup(*p_c*, G) , distribute(B) ,* S_B *, regroup(*p_d*, B) , distribute(T) ,* // T2 , B3
compute(c_a , SE_T*) , compute (*c_b , VE_B*) ,* S_G *, devolve(B) ,*
S_B *, regroup(*p_d*, B) , distribute(T) , compute(*c_a , SE_T*) , compute (*c_b , VE_B*)*

Figure 5.7: Rule Specialization and Composition (GPU Example): In addition to the triangles and circles introduced in the previous diagram, a cross represents vector compute according to the indicated codelet.

spectrum on a *single block* in a GPU. By applying $B3$, the `sum` is performed by the cooperative codelet. By applying $B1$ then $T2$, the entire `sum` is performed by a single thread in the block. By applying $B4$ then $T2$ and $B3$, the `sum` is distributed to the individual threads by the partition in codelet c, each thread then calculates partial sums, and then the partial sums are aggregated with the cooperative codelet. If instead of $T2$ and $B3$ we apply $T2$ and $B1$, followed by $T2$, then the partial sums are aggregated by a single thread in the thread block.

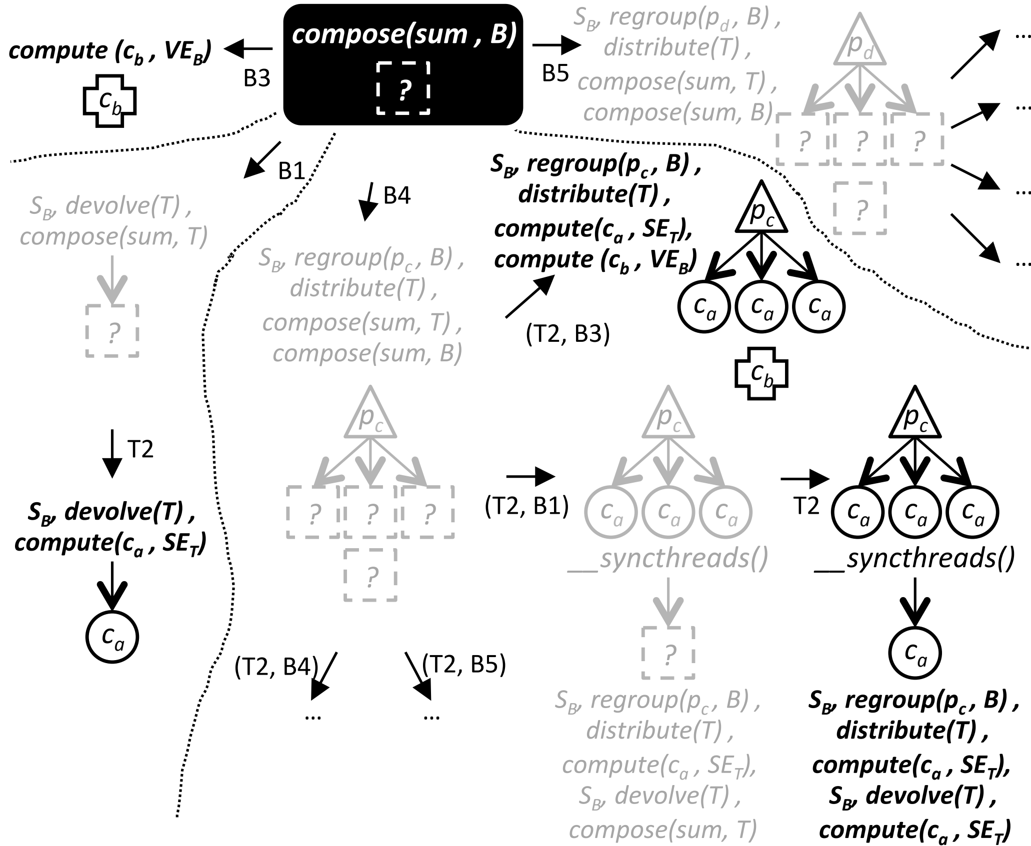


Figure 5.8: Example of Design Space for Composition Plans (Showing Four Possible Composition Plans for the Block and Thread Levels)

5.4 Model Extensibility

To keep the previous examples simple, we have modeled CPUs as two-level devices and GPUs as three-level devices. However, the architectural model is extensible as we show in the following examples.

CPU SIMD Unit. The SIMD (vector) unit of a CPU can be added by giving the thread level vector execution capability as well as a subordinate level consisting of a vector lane with scalar execution capability.

GPU Warp. Warp-centric mapping [71] on GPUs can be achieved by treating warps as a separate level between blocks and threads and giving the warp level a vector execution capability. Doing so enables more optimal code generation for cooperative codelets featuring warp-centric mapping optimization techniques. Such techniques include avoiding the use of `__syncthreads()` as well as using shuffle instructions and registers instead of scratchpad memory to store shared data.

Instruction-level Parallelism (ILP). On both CPUs and GPUs, ILP can be achieved via a subordinate level to the thread level that executes a serialized `map` loop that is unrolled. In this case, the subordinate level to the thread is the iteration of an unrolled loop and the synchronization happens by closing the loop.

GPU Dynamic Parallelism. Dynamic parallelism on GPUs can be achieved by assigning the grid level as a subordinate level to the thread level. This enables threads to decompose work and delegate to subordinate grids through new kernel launches which creates a cycle in the architecture hierarchy. Optimizations [46] involving dynamic parallelism can be considered alternative choices of composition.

A modern CPU typically is modeled as four-level devices (process, thread with SIMD unit, SIMD lane, and ILP) and a GPU is modeled as five-level devices (grid, block, warp, thread, and ILP). We leave support for dynamic parallelism for future work.

5.5 Discussion of Hardware Abstraction

Hardware abstraction is mainly proposed for modeling an architecture for its architectural hierarchy and further guiding a composition plan. Compared to conventional architectural models, our proposed model might be too "coarse-grained" and be deficient in finer architectural details, such as resource sizes, including core numbers, SIMD width, cache sizes, etc. However, as studied in Chapter 2, the architectural hierarchies mainly determine the algorithmic structure, while the resource sizes mainly impact the fine-grained optimizations, such as data placement and parameter tuning. These fine-grained optimizations will be discussed in the next chapters.

CHAPTER 6

TANGRAM COMPILER

In the previous chapters, we mentioned that both languages and compilers might impact performance portability, and discussed the relationship between the language and the corresponding design space. The TANGRAM language is proposed to deliver both high coverage and easy expression by adopting features of composition-based languages. We further discussed TANGRAM's composition mechanism with hardware abstraction. In this chapter, we first discuss the compiler factor for performance portability, and then detail the design and implementation of the TANGRAM compiler.

6.1 Compiler, Design Space and Performance Portability

Given a codebase written in a particular language, the design space discussed in Chapter 3 is determined, but different compilers might deliver different performance. As defined in Section 3.2, the design space S implies a set of all possible transformed versions among all possible compilers. Different compilers might have different transformation capabilities, select different transformation passes/paths, and eventually generate different output versions.

Two major kinds of issues can be observed here. First, **transformation** issues are defined as all possible transformation capabilities that impact performance, including (1) lack of one particular code transformation and (2) overheads introduced by one particular code transformation. Both cases can be considered as design space reduction. In the former case, the design space of compilation, denoted as S_C, can be considered as a set formed by sub-

Parts of this chapter appeared in the International Symposium on Microarchitecture [69]. The material is used with permission.

tracting all missing code transformations from S:

$$S_C = S - S_M,$$

where S_M is defined as all transformed versions requiring the missing code transformations. In the latter case, the versions with and without overheads, denoted as v_o and v, should be both in S due to the definition (a set of all possible transformed versions among all possible compilers). However, in this compiler, both are projected to v_o, so consequentially S_C only includes v_o. Therefore, it is also considered as design space reduction.

Second, **selection** issues are simply defined as suboptimal choices for the given design space of compilation, S_C. The selection issues typically belong to a research domain called *superoptimization*. The issues can come from various factors, including mismatched selection criteria, analyses, or assumption of target architectures. In the same design space S_C, different applied transformation passes and even different orders of transformation passes might generate different versions, delivering different performance.

These two kinds of issues are typically coupled together, and cannot be easily isolated in most compilers, because (1) most compilers might not be transparent in their transformation capabilities, making S_C unclear, or (2) most compilers deliver only one output version, coming with mixed effects of both kinds of issues. In the following two case studies, we are able to show examples for both kinds of issues.

Figure 6.1 shows different loop transformations of three different OpenCL compilers on an i7-3820 CPU. This is the same study discussed in Section 2.1.1 with both Parboil [20] and Rodinia [72] benchmarks. The figure only shows geometric mean, since the detailed numbers can be found in [19, 16] . Their loop transformations are documented in [26, 30, 19, 16], so it is more clear than other non-transparent compilers. The AMD compiler iterates in-kernel loops first over work-item loops (denoted as depth-first-order (DFO)), while the Intel compiler iterates work-item loops first in a vector width over in-kernel loops (denoted as breadth-first-order (BFO) within a vector). Lack of the other loop transformations might reduce the design space of compilation, consequentially delivering limited performance. Locality-centric scheduling proposed by Kim [19, 16] provides a compiler capability support for both DFO and BFO, further extending the design space

of compilation. On top of that, for selection issues, Kim also proposed an adaptive selection process by analyzing access patterns.

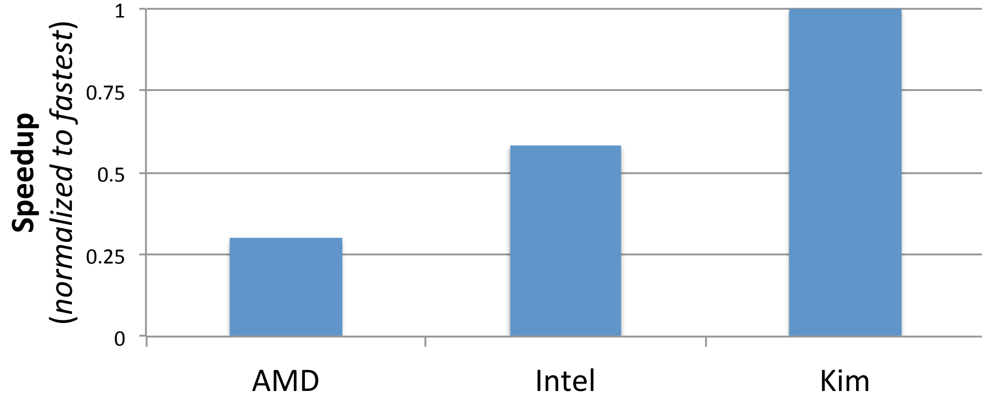


Figure 6.1: Performance Impact for Loop Transformations of OpenCL Compilers on an i7-3820 CPU

On the other hand, Figure 6.2 demonstrates different choices of the width in vectorization of the same Intel OpenCL compiler on the same CPU. The Intel OpenCL stack allows users to override the default heuristic by specifying the vector width as a compiler parameter. This study compares the performance of the heuristically selected vectorization [73] of `SGEMM` and `SpMV` in Parboil [20] against that of a non-vectorized version and two vectorized versions with different widths. The Intel compiler chooses 4-way vector for regular and control divergence free `SGEMM` kernel, while it uses 8-way vector for `SpMV` kernel which exercises control divergence. Under control divergence, code generation for work-items using SIMD instructions typically comes with a large overhead due to masking, packing and unpacking. In the end, Intel's heuristic has made suboptimal decisions for both cases, falling short of the best achieved performance by factors of 2.13x and 1.24x, respectively.

In this study, the design space of compilation should be the same and only the selection process is different due to using the same compiler. We conclude the selection issues exist in the Intel OpenCL compilers. We also note the Intel compiler does not use a fixed width for vectorization. That implies its heuristic might select the vector width based on some analyses.

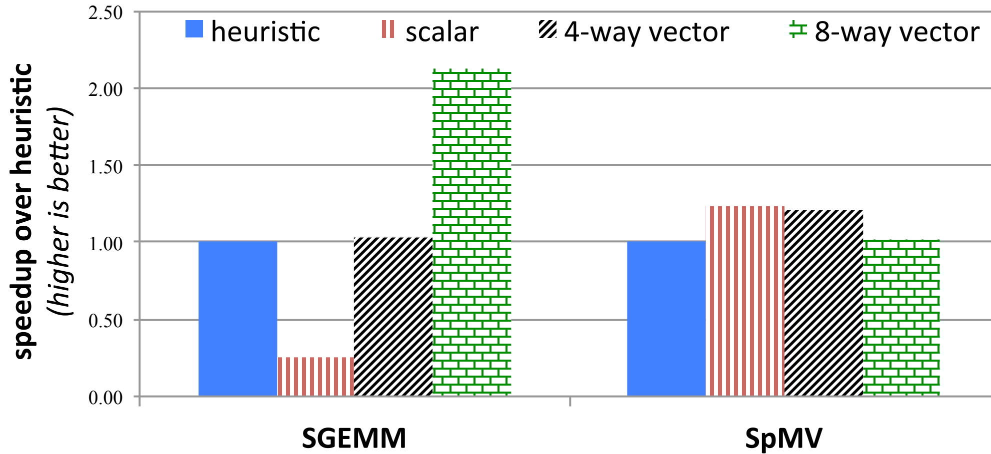


Figure 6.2: Performance Impact for Different Vectorization Strategies of Intel CPU OpenCL

6.2 TANGRAM Compiler Design Overview

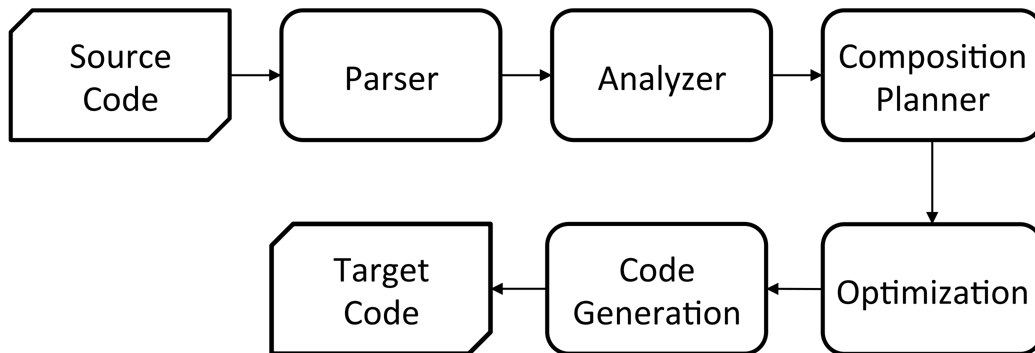


Figure 6.3: TANGRAM Compiler Organization

The TANGRAM compiler is designed as a source-to-source compiler to leverage sophisticated backend compilers. Figure 6.3 shows the organization of the TANGRAM compiler. In the compiler, the parser processes input code and forms TANGRAM abstract syntax tree (AST). Different from the ASTs in conventional languages, the TANGRAM AST introduces high-level nodes, such as spectrums, codelets, and primitives, and encodes high-level information, such as qualifiers, algorithmic choices, and scheduling policies.

The TANGRAM's intermediate representation (IR) is also the TANGRAM's AST. All of the analyses, transformations, including compositions and optimizations, and code generation across target devices rely on AST traversal. The analyzer traverses ASTs, diagnoses, and collects corresponding infor-

mation, such as access patterns or which codelets are compound or atomic. The composition planner traverses TANGRAM ASTs, and then constructs an AST for the output language with a determined algorithmic structure by driving the cloner. Finally, the optimization stage performs code optimizations on the composed AST, and the code generation stage generates the target code. Both optimization and code generation will be discussed in the next chapter

In the following sections, we will first introduce the TANGRAM infrastructure. Then we will discuss the detailed design and implementation of the first three stages in Figure 6.3 using the infrastructure.

6.3 TANGRAM Compiler Infrastructure

Similar to most compiler infrastructures [74, 75, 76], the TANGRAM infrastructure is designed to simplify and modularize the TANGRAM compiler. Different from other infrastructures, the TANGRAM infrastructure is specialized for compositions.

The TANGRAM infrastructure is implemented in C++ with an object-oriented design. It makes the compiler extensible for new language features, analyses, composition heuristics, compositions, optimizations, and output languages. The infrastructure includes two core components, the TANGRAM AST nodes and the AST visitors (or traversers). The former components encode high-level information crucial for compositions, while the latter components form compositions themselves as well as optimizations and code generation.

6.3.1 TANGRAM AST

In the TANGRAM compiler infrastructure, a unified set of AST nodes is used for both the TANGRAM language and output languages (OpenCL, CUDA, or OpenMP). This unified AST is called the TANGRAM AST.

This AST adopts specialized, unique, high-level nodes to represent spectrums, codelets, primitives, and special functions. These specialized types of nodes are applied only for the TANGRAM language, not for output languages. The TANGRAM AST also encodes high-level information, such as

qualifiers, algorithmic choices, and scheduling policies in its nodes.

It is almost one-to-one mapping between the TANGRAM objects (Section 4.2) and TANGRAM-specialized AST nodes. The corresponding composition rules are discussed in Chapter 5. The *spectrum call* node and the `map` node are the most crucial nodes. The former is specialized as a kernel or a function to support functional polymorphism, while the latter is specialized for a proper scheduling policy to support nested parallelism. Both of these processes happen in the composition planner. Those nodes eventually generate different output code with different specialization in the code generation stage. For example, a `map` node might be specialized with different scheduling policies, such as Distribute or Serialize (Section 5.2.1). Depending on its policies, it might generate a serial `for` loop, a `for` with parallel pragma in OpenMP, or a hidden loop like thread or thread block in CUDA.

Moreover, the `partition` node and the *subscript* node for a TANGRAM's `Array` container are also crucial for achieving good locality. They typically are analyzed in the analyzer and transformed in the optimization or code generation stage. Special functions, such as `coopIdx()` and `coopDim()` in cooperative codelets and `size()` in `Array` containers, are converted in common nodes in the code generation.

Each node is implemented as a C++ class, and inherited properly based on the grammar. This object-oriented design makes the infrastructure extensible for new language features.

6.3.2 AST Visitor

In terms of design for transformations, the TANGRAM infrastructure takes a similar design philosophy as Clang [74]. Each analysis, transformation, or code generation can be one *pass* of AST traversal or can be decomposed into multiple passes.

For each node, it has its own traversing function with prefix, infix and postfix operations. Also prefix, infix and postfix operations all have base operations, which can be called across all AST nodes. All of the above functions or operations are implemented as virtual functions and can be overridden. A default traversing function provides generic traversal based on its type. An optional auxiliary object as a parameter of the traversing function is also

provided. Furthermore, the return objects of these functions contain multiple common data structures for compiler analyses or transformations, and they also can be overridden.

Algorithm 1 shows a generic pattern of a visitor for a `TNode`, and related locations of prefix, infix, and postfix operations. It also shows return objects (*RO* and *newRO*) and optional auxiliary objects (*Aux* and *newAux*). In the following discussion, we will remove optional auxiliary objects, if they are not used.

Algorithm 1 Generic Visitor

```
1: procedure TRAVERSETNODE(TNode,Aux)            ▷ traversing function
2:     preOpTNode(...)                           ▷ prefix operation
3:     for all c's in TNode's childrenList do
4:         RO ← traverse(c, newAux)
5:         inOpTNode(...)                        ▷ infix operation
6:     end for
7:     postOpTNode(...)                          ▷ postfix operation
8:     return newRO                              ▷ return object
9: end procedure
```

The infrastructure provides multiple templates of AST visitor (or traversers). The compiler designer only needs to write a pass by inheriting a proper visitor template and providing minimum modification, which could include (1) overriding traversing functions, or prefix/posfix operations for nodes of interest (NOI), (2) specifying the base operations for all traversing functions, and/or (3) using or overriding return object classes for all traversing functions. Several AST visitors have read-only input ASTs and construct output ASTs with modified information if there are output ASTs They mostly operate on TANGRAM-specialized ASTs. A few AST visitors allow in-place AST modification, and typically happen in the code generation stage.

Since AST visitors are widely used in the TANGRAM compiler, this object-oriented design of the AST visitors can make the infrastructure extensible for new analyses, compositions, optimizations, and output languages. The detailed discussion is given in the next sections. Passes related to optimization or code generation is discussed in the next chapter.

6.4 Parser

The TANGRAM language is implemented as an extension of C++. We rely on Clang to parse TANGRAM code. More specifically, containers, special functions, and primitives are implemented as C++11 template classes, which can be easily parsed by Clang. This design makes the TANGRAM language extensible.

On the other hand, TANGRAM's qualifiers are implemented as keywords, which require modifying Clang. Potentially, the qualifiers can be implemented as C++ customized attributes, avoiding modifying Clang. We leave this for future work.

After TANGRAM code parsed by Clang, the Clang AST is constructed. We apply the Clang AST visitor to traverse the Clang AST, and construct the TANGRAM AST. By modifying Clang to support TANGRAM qualifiers, we can easily identify spectrums and codelets for the parser. All of the remaining qualifiers, such as `__shared` or `__tunable`, are stored in the corresponding AST node with the labeled expression. The parser also maintains a lookup table to record all spectrums for quick accesses. For primitives and special functions, we parse them based on class names or function names. We currently support most C features, except for `struct` and `union`.

6.5 Analyzer

The TANGRAM analyzer is an AST visitor that traverses a read-only TANGRAM AST, analyzes spectrums and codelets, and builds an information lookup table for the spectrums and codelets.

Codelet properties are the most critical factors for composition to minimize selection issues of compilers. Here, we focuses on two major kinds of properties: **parallelism** and **locality**.

After collecting all properties, a process of *level matching* is performed by binding all the codelets with each level of hardware abstraction based on their properties and its capability. This step can provide a quick access to each applicable codelet for each level in the later steps. Its process is very similar to composition specialization (Section 5.3) but without deriving rules.

6.5.1 Parallelism

The parallelism properties are mainly determined by types of codelets, which are determined by traversing and analyzing their bodies. A compound codelet with `map` primitives generally has more parallelism than others. While an autonomous codelet has no parallelism, a cooperative codelet also has a degree of parallelism but for a level with vector execution capability. For a compound codelet without a `map` primitive, its parallelism is determined by its callee spectrum.

Algorithm 2 Analyzer

```
1: Lookup(...).converged ← false
2: procedure ANALYZER(SList)            ▷ SList is a list for all spectrums
3:     for SList iterations do
4:         for all St's in SList do
5:             if Lookup(S).converged = false then
6:                 traverse(S)
7:             end if
8:         end for
9:     end for
10: end procedure
```

On the other hand, parallelism of a spectrum is determined by all of its codelets. Since a codelet can be recursive by calling a spectrum not analyzed yet, it makes this analysis iterative until convergence . Each spectrum can be considered as a vertex of a graph, and a caller/callee relationship between two spectrums can be viewed as a directed edge. This algorithm is a brute-force algorithm for graph traverse to form a spanning tree, and has an upper bound of a $O(N^2)$ complexity to converge, where N is the number of spectrums. Each vertex converges only when all sink vertices of its all outgoing edges converge. In this brute-force algorithm, vertices are traversed with a fixed order, which is a declaration order in the input TANGRAM code. In each step of traversing a spectrum (a vertex), all codelets of this spectrum are traversed and checked for convergence. A codelet with a spectrum call (a edge) converges only when the called spectrum converges. Here, we can check only unconverged spectrums and codelets to reduce the complexity. It is trivial that the worst case happens at the forward-order traversal. Algorithm 2 shows pseudo code discussed above. Here, *Lookup* stores information for spectrums and codelets, and *traverse* is a generic function interface for

all traversing functions.

Additional to checking convergence, each step of traversal also collects properties of codelets and spectrums as follows:

1. A cooperative property can be determined by checking its `__coop` qualifier.

2. A compound property can de determined by checking existence of a callee spectrum.

3. A `map` property can be determined by checking existence of a `map` primitive.

4. An autonomous property can be determined by not satisfying 1 to 3.

5. Properties can be propagated through a callee spectrum in a compound codelet.

Properties of a spectrum are also determined by all of its codelets. These properties are finalized only when a spectrum or a codelet has converged.

This analyzer is implemented using a visitor template in the TANGRAM infrastructure. This template is designed to traverse TANGRAM AST with recursive spectrums and codelets. It presets the converge property for each AST node based on the grammar. For example, in an identifier node, it checks whether the identifier points to a TANGRAM-related node, and uses that to set the convergence.

The analyzer first inherits the template and overrides the traversing functions or prefix/postfix operations for spectrum, codelet, spectrum call, and `map` AST nodes. Algorithm 3 shows pseudocode of an example that detects a `map` primitive. Here, $traverseSpectrum$ and $traverseCodelet$ are the traversing functions specific for spectrum and codelet nodes respectively, $preOpMap$ is the prefix operation function for `map` primitives, and $inOpBase$ is the base infix operation function for all nodes.

We simply modify $traverseSpectrum$ and $traverseCodelet$ to store information for detecting `map` primitives. Here, we show the detailed code for $traverseSpectrum$ for both convergence, and `map` detection. While the convergence property requires all children nodes to be converged, the `map` property requires at least one child node to have a `map`. In both $traverseSpectrum$

Algorithm 3 Map Detection

```
1:  Lookup(...).converged ← false
2:  Lookup(...).hasMap ← false
3:  procedure traverseSpectrum(S)
4:      AllConverge ← true
5:      for all codelet C's in S do
6:          if Lookup(C).converged = false then
7:              RO ← traverse(C)
8:              AllCoverage ← AllCoverage AND RO.getInfo(converged)
9:          end if
10:     end for
11:     if AllConverge = true then
12:         Lookup(S).converged ← AllConverge
13:         OneMap ← false
14:         for all codelet C's in S do
15:             OneMap ← OneMap OR Lookup(C).hasMap
16:         end for
17:         Lookup(S).hasMap ← OneMap
18:     end if
19:     NewRO.addInfo(converged) ← AllCoverage
20:     return NewRO
21: end procedure
22:
23: procedure traverseCodelet(C)
24:     ...
25:     NewRO.addInfo(converged) ← ...                          ▷ Original code
26:     NewRO.addInfo(converged) ← ...                          ▷ Original code
27:     Lookup(C).converged ← NewRO.getInfo(converged)          ▷ New code
28:     Lookup(C).hasMap ← NewRO.getInfo(hasMap)                ▷ New code
29:     return NewRO
30: end procedure
31:
32: procedure preOpMap(M)
33:     NewRO.addInfo(hasMap) ← true
34:     return NewRO
35: end procedure
36:
37: procedure inOpBase(RO1, RO2)                                ▷ Merge all RO info
38:     ...
39:             ▷ Add a logic to merge RO.hasMap, which is an OR operation
40:     ...
41: end procedure
```

and *traverseCodelet*, we update *Lookup* to keep information and avoid redundant traversal. Then we override *preOpMap* by setting *hasMap* to true, and add the logic for merging *hasMap* (the `map` property), by checking at least one return object from children nodes to have a *hasMap* set as true.

6.5.2 Locality

Similar to parallelism, locality also requires codelets and spectrums to be converged during the analysis. Different from parallelism, locality is determined by checking access patterns, including `partition` primitives, corresponding `sequence` primitives, and *subscript indices* of `Array` containers. Currently, an access is determined as zero stride (constant), unit stride, or non-unit regular stride, and irregular stride. The analysis algorithm for locality is very similar to those for parallelism. It is implemented using the same analyzer template but with different overridden functions for AST nodes. In this analysis, the traversing functions for spectrum, codelet, spectrum call, `partition`, `sequence`, and subscript index of `Array` container nodes are overridden. For brevity and clarity, the pseudocode is omitted. Since locality and parallelism are independent, both analyses can be merged in one pass.

6.5.3 Level Matching

Table 6.1: Relationship between Level Capability to Codelet Property

Codelet Property		**autonomous**	**cooperative**	`map`
Level Capability	**scalar execution**	✓	✗	✓
	vector execution	✗	✓	✗
	distribute capability	✗	✗	✓

The process of *level matching* binds all the codelets with each level of hardware abstraction based on the codelets' properties and the level's capability. Table 6.1 summarizes the mapping between the level capabilities and the codelet properties. The scalar execution can be matched to the autonomous and `map` properties, while the vector execution can be matched to the cooperative property. The distribute capability can be matched to the

`map` property. For each level, we can classify corresponding codelets based on these three capabilities, and construct a lookup table for quick accesses.

6.6 Composition Planner

The TANGRAM composition planner is a key step of the TANGRAM compiler. It handles compositions discussed in Chapter 5, and contains two components: an *AST cloner* to perform composition rules and a *heuristic* to select composition rules. In practice, the AST cloner is universal across architectures or objectives (e.g. performance or energy), while the heuristic might vary across architectures or objectives. In this document, we propose a single heuristic attempting to maximize performance across different architectures.

In the composition planner, it begins by composing the spectrum at the top level in the architectural hierarchy and then proceeds to explore the design space (like Figure 5.8) via a *breadth-first* search. At each point in the search space, if the candidate has no more composition rules to expand, it is considered complete and is passed to the next iteration as is. Otherwise, the algorithm selects the set of beneficial rules (*through the heuristic*) to apply at the corresponding objects and generates a new candidate for every combination of rules (*through the AST cloner*). This search process iterates for N iterations where N must be at least the number of architectural levels in order for the composition plans to reach the lowest level. Typically, N is a bit larger to enable a wider search.

6.6.1 AST Cloner

The AST cloner is an AST visitor that traverses a read-only TANGRAM AST and constructs another AST. The new AST is a copy of the original input AST with only one extra specialized node, which can be a spectrum call or a `map` node.

The cloner is implemented using a visitor template that *deeply clones* every node, and then overriding the traversing functions or operations for spectrum call, codelet, and `map` AST nodes. Given a spectrum call node, the cloner takes a parameter specifying which codelet is applied in this spectrum call

from the composition planner, then deeply clones that codelet, and finally appends the new copy of the codelet node with the newly constructed spectrum call node. Meanwhile, the cloner also searches the next unspecialized node, which is either an unspecialized spectrum call or an unspecialized `map` node, and returns it to the composition planner. Since there might be multiple candidates, all returned unspecialized nodes form the frontier for the next iteration of the design space search. Therefore, it is a breadth-first search. Similarly, given a `map` node, the cloner also takes a parameter specifying a scheduling policy for this `map` from the composition planner, then deeply clones the `map` node, and finally writes this information in the new copy of the `map` node.

Algorithm 4 shows pseudocode for the AST cloner that only considers specialization of spectrum call nodes and only a single candidate for brevity and clarity. A pointer *UnSpecialized* is applied to track the unspecialized node, so the planner can use it to quickly access the analyzed results. Auxiliary objects (*Aux* and *newAux*) are used to control the cloning phase and the search phase of a spectrum call node. For a node other than a spectrum call node, *newAux* is set as *Aux* in *preOpBase*, enabling deep search. The algorithm simply specializes the node with *isSpecializing*() equal to true, and searches the next unspecialized node in each iteration.

This AST specialization process can be considered as applying one single composition rule (Section 5.2) to that extra specialized node. As mentioned, each codelet typically implies a program composition rule. Slightly different from Sections 5.2.2 and 5.3, where we illustrate that rules are first derived together and then specialized for levels for brevity and clarity, in the TANGRAM compiler, we implement compositions by deriving and specializing each rule at the moment when the rule is applied for a level. The former method can be considered as *pre-deriving* the rules with *memorization* to avoid re-deriving, while the latter can be considered *lazy* deriving and re-deriving. Although the former method seems to save computation by a tradeoff of memory, in practice, the latter provides a much simpler compiler design. Also, given an architecture and an application, not every rule is eventually applied in a composition plan, so there is no need to derive all rules in the beginning.

Algorithm 4 AST Cloner

```
1: SClocal ← the top spectrum call (SCtop)
2: UnSpecialized ← SClocal
3: SClocal.isSpecializing(true)
4: procedure CompositionPlanner( )        ▷ Only for caller's behavor
5:     ...
6:     for ... do
7:         ...
8:         check analyzed results (in the Analyzer) of UnSpecialized
9:         ...
10:        Aux.addParameter(Choice)        ▷ Heuristic select a codetet
11:        RO ← traverse(SClocal, Aux)        ▷ Specialization
12:        ...
13:        SClocal = RO.getNode(SClocal)
14:        ...
15:        Aux.addBool(true)        ▷ Enable search
16:        RO ← traverse(SClocal, Aux)        ▷ Search unspecialized node
17:        ...
18:        SClocal = RO.getNode(SClocal)
19:        ...
20:    end for
21:    ...
22: end procedure
23: procedure traverseSpectrumCall(SC,Aux)
24:    ...        ▷ Remove preOpSpectrumCall
25:    if Aux.getBool() then        ▷ Search unspecialized node
26:        UnSpecialized ← SC
27:        UnSpecialized.isSpecializing(true)
28:    end if
29:    ...        ▷ Original code (cloning everything)
30:    if SC.isSpecialzing() then
31:        UnSpecialized ← Null        ▷ Reset
32:        Choice ← Aux.getParameter()        ▷ Set by planner
33:        RO ← traverse(SC.getSpectrum().getCodelet(Choice), newAux)
34:        NewSC.setCodelet(RO.getNode())
35:        NewSC.isSpecialzing(false)
36:        NewSC.isSpecialzed(true)
37:    end if
38:    newRO.setNode(NewSC)        ▷ Original code
39:    return newRO
40: end procedure
41: procedure preOpBase(Aux, newAux)
42:    newAux ← Aux
43: end procedure
```

6.6.2 Heuristic for Composition Plan

```
s0                     is the spectrum subject to kernel synthesis
L                      is the top level in the device being targeted
N                      is the number of iterations to search
candidates(i)          is the set of composition candidates at iteration i
rules(s, ℓ)            is the set specialized rules for spectrum s at level ℓ
prune(rules, i)        sorts and prunes rules for iteration i
prune(candidates, i)   sorts and prunes candidates for iteration i

candidates(0) := { compose(s0, L) }
for iteration i from 1 to N do
  forall c ∈ candidates(i-1) do
    if no calls compose(s, ℓ) in c then
      candidates(i) ← c      // propagate complete candidates
    else
      forall compose(s, ℓ) in c do
        forall r in prune(rules(s, ℓ), i) do
          mark r as a candidate rule for compose(s, ℓ) in c
      B := all combinations of candidate rules for c
      forall b in B do
        candidates(i) ← c with all rules in b applied
  candidates(i) := prune(candidates(i), i)
```

Figure 6.4: Composition Algorithm with Pruning

To avoid explosion of the search space, *pruning* takes place throughout the process when specialized rules are being selected as well as in between iterations. Figure 6.4 shows the brief algorithm for the composition planner with pruning. It is still the breadth-first search algorithm we discussed before, but with a `prune` function that sorts rules or candidates according to their expected benefit, and then drops the lowest ones. The strictness of pruning can be set by the user and determines how many candidates to keep or drop.

In the TANGRAM compiler, the pruning policy currently used is *parallelism first* whereby rules extracting more parallelism are prioritized. Here parallelism is determined by the TANGRAM analyzer (Section 6.5). The criteria for comparing two rules according to their benefit is shown in Figure 6.5. Note that we only consider applicable rules as the compared rules (r_1 and r_2) for the level (ℓ). The applicable rules per level can be accessed through the lookup table constructed in the level matching.

If we are not in the last iteration, rules that generate *distribute* capability are preferred because they extract more parallelism. Among rules that dis-

```
compare(r1, r2, ℓ, i):
    # Comparing rules r1 and r2 (of level ℓ) for composing at iteration i
    if i is not the last iteration then
        prefer(distributes)
        if r1 distributes and r2 distributes then
            prefer(partitioning matches ℓ)
        else        # neither distributes
            prefer(vectorizes)
    else            # i is the last iteration (distribute is undesirable)
        prefer(computes)
        if r1 computes and r2 computes then
            prefer(vectorizes)
    prefer(came from tunable codelet)
    return both are the same

prefer(cond):  if one rule satisfies cond and the other doesn't, return the
               rule that does, otherwise continue with the execution
```

Figure 6.5: Comparing Composition Rules for Pruning

tribute, the partitioning schemes are analyzed and used to determine which rule has more favorable *locality* for the level in question. The preference of the level is determined by the architecture specification via an additional entry for each level that specifies whether it prefers adjacent or strided tiling. If neither rule distributes, then rules generating vector execution are preferred over those generating scalar execution. In the last iteration, rules that compute are preferred over those that distribute because there will be no more iterations to expand the distributed *compose* invocations. Among compute rules, those that generate vector execution are preferred. When all is equal, rules that come from codelets having tuning knobs are preferred because they give the compiler more optimization opportunities. Candidates' plans are compared via pairwise comparison of the rules applied to each.

CHAPTER 7

OPTIMIZATION AND CODE GENERATION

In the previous chapter, we discussed how the composition planner is designed to specialize the algorithmic structure for a spectrum call and the scheduling policy for `map` at a level of hierarchy on a particular architecture. However, given an algorithmic structure for a function, performance could still vary significantly due to lack of optimizations. Depending on the (output) languages, the corresponding compilers and even the target device, optimizations might be necessary for gaining performance. For example, OpenCL, as a low-level language, typically requires optimizations to be expressed. Particularly for GPU, data placement is crucial for performance, but is not included as an automatic transformation in most OpenCL compilers.

The TANGRAM compiler therefore includes multiple high-level optimizations for performance portability to bridge the performance gap of the backend languages and compilers. Also the TANGRAM language forces programmers to encode high-level information, such as the qualifiers and primitives, which can facilitate multiple optimizations related to memory and parameter tuning (Section 4.2).

In the TANGRAM compiler, the optimization stage typically includes multiple optimizers, each of which traverses an AST, applies corresponding optimization, and then constructs another AST. The code generation stage, also including multiple AST visitors, traverses the optimized AST and generates output code in an output language (e.g. OpenCL, CUDA, or OpenMP). In practice, the above two stages vary across types of architectures, languages, or objectives.

Parts of this chapter appeared in the International Symposium on Microarchitecture [69]. The material is used with permission.

7.1 Optimization

The current TANGRAM optimization stage includes two optimizers, one for *data placement* and one for *parameterization*. The data placement optimizer assigns data to proper memory spaces, such as scratchpad memory, registers, texture memory, and global memory. Data placement is crucial for high performance because on-chip memory can deliver significantly more bandwidth than global memory. However, data might not be always suitable for on-chip memory, due to its access pattern. The parameterization optimizer collects tunable variables, and then either assigns proper values or templatizes the synthesized code with parameters. Parameterization is important for performance tuning to improve resource utilization.

In general, an optimizer is an AST visitor that traverses an AST, then applies the optimization, and finally either constructs a new AST or directly modifies the old AST. In these two optimizers, we applied the latter method.

7.1.1 Data Placement

The data placement optimizer is implemented using a blank AST visitor that simply traverses an AST without any prefix, postfix, or infix operation, and then overrides the traversing functions for `Array` container, and corresponding subscript index AST nodes to perform AST modification.

Figure 7.1 shows the heuristic algorithm used in the current data placement optimizer particularly for GPUs. The information of the `__shared` qualifier is captured in the parser (Section 6.4) and then stored in `Array` container nodes, and the stride information is also available in the TANGRAM analyzer (Section 6.5).

If a container is shared, it is considered for on-chip caching. Otherwise, the stride information from the analyzer is used to determine data placement. If the stride is zero (i.e., invariant with respect to map elements) or constant (greater than one), the containers are also considered for on-chip caching. The constant-strided container might need a transpose on scratchpad memory. If the stride is a unit then the container is placed in the default global memory. Hardware is already well-designed to handle stride-one accesses (e.g., CPU caches, GPU coalescing). If the stride is irregular, the container is placed in the cached global memory or texture memory.

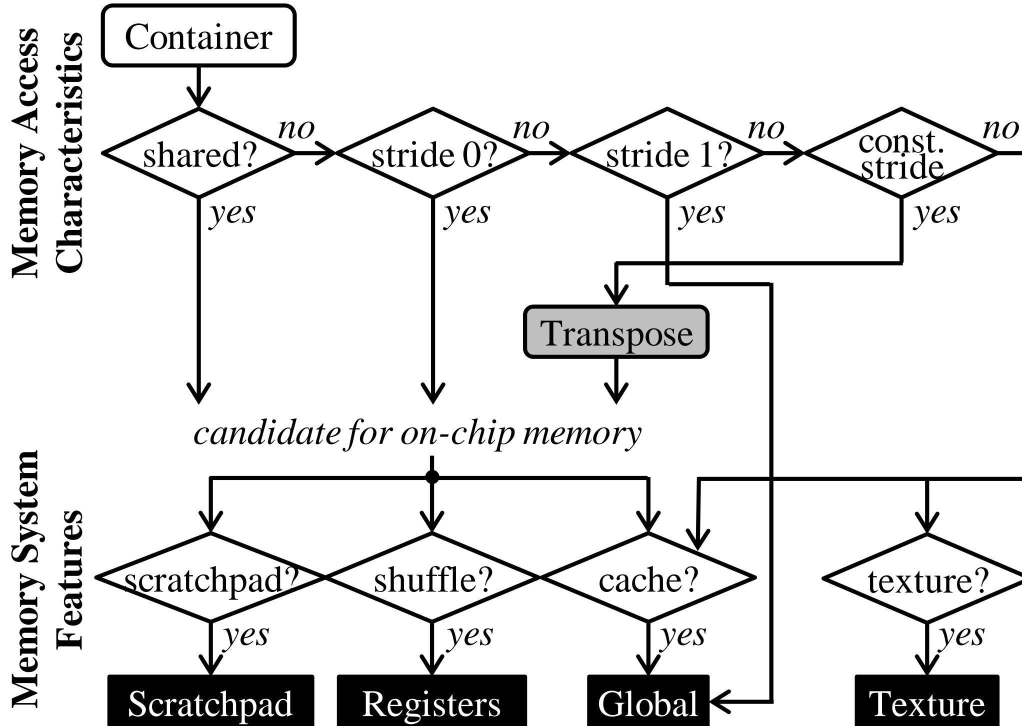


Figure 7.1: Heuristic for Data Placement

7.1.2 Parameterization

The parameterization optimizer is implemented using a collective AST visitor template that collects a list by merging corresponding lists of children nodes. We then override the traversing function of declaration nodes to enqueue, and the traversing functions of spectrum call nodes to modify the corresponding ASTs. The modification appends the collected nodes to a spectrum call node to enable templatization. The information of the `__tunable` qualifier is stored in declaration nodes. Therefore, it is trivial to recognize tunable variables.

Algorithm 5 briefly shows the mechanism of the parameterization optimizer. For each declaration node with a tunable qualifier, we simply create a new identifier, set the identifier as the initializer of the declaration, and collect the identifiers. For each spectrum call node, we collect its descendant parameterized identifiers.

Algorithm 5 Parameterization

```
 1: procedure traverseSpectrumCall(SC)
 2:     ...                                              ▷ Original code
 3:     for all node N's in newRO.getNodeList() do
 4:         SC.addTemplateParameter(N)
 5:     end for
 6:     return newRO
 7: end procedure
 8: procedure traverseDeclare(D)
 9:     if D.isTunable() then
10:         newID ← new Identifier()
11:         D.setInitializer(newID)
12:         newRO.addNode(newID)
13:     end if
14:     return newRO
15: end procedure
```

7.2 Code Generation

The current code generation stage has multiple preprocessors and one output code generator. Each preprocessor behaves like an optimizer, and then either targets at a small scope of AST modification, or collects specific information. Different from an optimizer, which is optional and applied for performance, a preprocessor is either required for correctness in code generation or applied to simplify the code generation stage. The output code generator simply prints out the AST in an output language.

7.2.1 Codegen Preprocessors

Table 7.1 lists the functionality of each preprocessor, its corresponding overridden traversing functions, and its properties. Since they are all similar to an optimizer but with a different degree of modification in overridden functions, we omit the explanation for brevity. Some preprocessors are not related to any output language, so they are universal across output languages. Some preprocessors are specific for one output language, since they are designed to adjust AST for satisfying the output language.

The current preprocessors can be classified into two categories in terms of implementing mechanisms:

Table 7.1: Codegen Preprocessors

Preprocessors	Language*	Description
Index calculation	U	Calculate and linearize subscript indices
Address alignment	U	Calculate address alignment and offset for a container
Special function	C,G	Convert special function AST nodes to ouput AST nodes
Return promotion	U	Promote return nodes to parameter nodes and arguments
Name mangling	U	Perform name mangling for identifier nodes
Function collection	U	Collect synthesized kernels, device functions, or functions
Qualifier preparation	C,G	Identify and apply qualifiers for kernels, device functions, or inline functions
Kernel configuration	G	Add OpenCL or CUDA configuration nodes

U: Universal. G: CUDA or OpenCL. C: OpenMP.

1. Modify and collect AST nodes. This has an implementing mechanism similar to that of the parameterization optimizer.

2. Modify AST nodes without collecting nodes. This is similar to the data placement optimizer.

If multiple preprocessors are independent, they potentially can be merged and implemented as a larger AST visitor in order to traverse the AST fewer times. However, that might sacrifice the clarity of the compiler implementation. Therefore, we do not apply merging. Also, a preprocessor might depend on another or require another as a backend, so the order of preprocessors might be crucial. For example, in our implementation, index calculation needs a special function preprocessor as a backend, because our implementation for the index calculation preprocessor might insert special functions into the AST. Those special functions are required to convert to output ASTs in a special function preprocessor.

7.2.2 Output Code Generator

In the TANGRAM design philosophy, we tend to keep the output code generator as simple as possible by moving sophisticated operations to the preprocessors. The output code generator is implemented using a full customized AST visitor, which means all of the traversing functions are overridden. The code generator traverses a read-only AST and prints out the output code. Each output language typically has its own code generator. Figures 7.2 and 7.3 show two examples for code generation in the simple CPU and GPU

S_P, regroup(p_c, P), distribute(T), compute(c_a, SE_T),
S_P, devolve(T), compute(c_a, SE_T)

```
S_P              :  // No sync needed at the beginning
regroup(p_c, P)  :  unsigned p_c = omp_get_num_threads();
regroup(p_c, P)  :  unsigned len_c = in_size;
regroup(p_c, P)  :  unsigned tile_c = (len_c+p-1)/p_c;
distribute(T)    :  #pragma omp parallel
distribute(T)    :  {
distribute(T)    :    unsigned j = omp_get_thread_num();
compute(c_a, SE_T) :    unsigned len_a1 = tile;
compute(c_a, SE_T) :    int accum_a1 = 0;
compute(c_a, SE_T) :    for(int i = 0; i < len_a1; ++i) {
compute(c_a, SE_T) :      accum_a1 += in[j*tile_c + i];
compute(c_a, SE_T) :    }
compute(c_a, SE_T) :    ret_a1[j] = accum_a1;
S_P              :  } // Join omp threads
devolve(T)       :  // No spawn (only master executes)
compute(c_a, SE_T) :  unsigned len_a2 = p;
compute(c_a, SE_T) :  int accum_a2 = 0;
compute(c_a, SE_T) :  for(int i = 0; i < len_a2; ++i)
compute(c_a, SE_T) :    accum_a2 += ret_a1[i];
compute(c_a, SE_T) :  ret_a2 = accum_a2;
```

Figure 7.2: Codegen for CPU Example in Figure 5.6

in Chapter 5, using OpenMP and CUDA as the output languages, respectively. Here, we use the notation of composition rules instead of specialized TANGRAM ASTs, since the full AST might not easily fit in a page. At mentioned, composition rules and specialized TANGRAM AST nodes are almost a one-to-one mapping (Sections 5.2 and 5.3).

S_G, *regroup*(p_c, *G*), *distribute*(*B*), S_B, *regroup*(p_d, *B*), *distribute*(*T*), *compute*(c_a, SE_T), *compute*(c_b, VE_B) S_G, *devolve*(*B*), S_B, *regroup*(p_d, *B*), *distribute*(*T*), *compute*(c_a, SE_T), *compute*(c_b, VE_B)

First kernel

```
S_G                :  // No sync needed at beginning
regroup(p_c, G)    :  unsigned p_c = gridDim.x;
regroup(p_c, G)    :  unsigned len_c = in_size;
regroup(p_c, G)    :  unsigned tile_c = (len_c+p_c-1)/p_c;
distribute(B)      :  unsigned k = blockIdx.x;
S_B                :  // No sync needed at beginning
regroup(p_d, B)    :  unsigned p_d = blockDim.x;
regroup(p_d, B)    :  unsigned len_d = tile_c;
regroup(p_d, B)    :  unsigned tile_d = (len_d+p_d-1)/p_d;
distribute(T)      :  unsigned j = threadIdx.x;
compute(c_a, SE_T) :  unsigned len_a = tile_d;
compute(c_a, SE_T) :  int accum_a = 0;
compute(c_a, SE_T) :  for(unsigned i=0; i < len_a; ++i) {
compute(c_a, SE_T) :    accum_a += in[k*tile_c + j + p_d*i];
compute(c_a, SE_T) :  }
compute(c_a, SE_T) :  ret_a = accum_a;
compute(c_b, VE_B) :  __shared__ int tmp[blockDim.x];
compute(c_b, VE_B) :  unsigned len_b = p_d;
compute(c_b, VE_B) :  unsigned id = threadIdx.x;
compute(c_b, VE_B) :  tmp[id] = ret_a;
compute(c_b, VE_B) :  __syncthreads();
compute(c_b, VE_B) :  for(unsigned s=1; s<blockDim.x; s *= 2) {
compute(c_b, VE_B) :    if(id >= s)
compute(c_b, VE_B) :      tmp[id] += tmp[id - s];
compute(c_b, VE_B) :    __syncthreads();
compute(c_b, VE_B) :  }
compute(c_b, VE_B) :  ret_b[k] = tmp[blockDim.x-1];
S_G                :  return; // Terminate kernel
```

Second kernel

```
devolve(B)         :  if(blockIdx.x == 0)
S_B until end      :      ... // Similar to first kernel
```

Figure 7.3: Codegen for GPU Example in Figure 5.7

CHAPTER 8

EVALUATION

The performance of the proposed system is evaluated in this chapter. We show that TANGRAM can deliver performance comparable to that of highly-optimized libraries.

8.1 Setup

The TANGRAM language is implemented as an extension of C++. The TANGRAM compiler is described in Chapters 6 and 7 with Clang [74] 3.6. Here we support C with OpenMP, and CUDA as the output languages. The generated kernels are then compiled using the Intel C compiler (icc) version 16.0.0, OpenMP version 4.0, and the NVIDIA CUDA compiler (nvcc) version 7.0 respectively. The compiled programs are evaluated on an i7-3820 Sandy Bridge CPU, a C2050 Fermi GPU, and a K20c Kepler GPU.

8.1.1 Benchmarks

Table 8.1 summarizes the applications implemented in TANGRAM: `Scan`, `SGEMV` (with 2 datasets, called TS and SF), `DGEMM`, `SpMV`, `KMeans`, and `BFS`, and the corresponding datasets and numbers of codelets. In selecting the benchmarks, we chose benchmarks associated with most impactful modern applications and meanwhile with highly optimized references. `Scan` is a primitive widely applied in relational algebra and data manipulation; `SGEMV`, `DGEMM` and `SpMV` are the most common tensor operators across modern applications, including deep learning; `KMeans` is a key algorithm of data analytics and machine learning; `BFS` is the most important graph traversal.

Parts of this chapter appeared in the International Symposium on Microarchitecture [69]. The material is used with permission.

Table 8.1: Benchmarks

Benchmark	Reference	Dataset	Number of Codelets (Input Code)
Scan	Thrust	A 16M integer array	4, with 2 exclusive scan and 4 reduction
SGEMV-TS	MKL & CUBLAS	A 512K-by-128 (Tall-and-Skinny) matrix	1, with 1 dot-product, and 2 reduction
SGEMV-SF	MKL & CUBLAS	A 128-by-512K (Short-and-fat) matrix	1, with 1 dot-product, and 2 reduction
DGEMM	MKL & CUBLAS	A non-transposed 4K-by-4K matrix & a transposed 4K-by-4K matrix	1, with 2 dot-product and 1 reduction
SpMV	MKL & CUSPARSE	`bcsstk18` [77] (CSR format)	1, with 1 sparse scalar-multiply, 1 sparse dot-product, and 4 reduction
KMeans	Rodinia	`kdd_cup` (default in Rodinia)	1, with 1 difference, 4 reduction, 1 minima selection, and 1 gemm-like operation
BFS	Rodinia	`graph1MW_6` (default in Rodinia)	1, with 1 edge visiting, and 1 vertex visiting

We compare each of our generated kernels to a reference, or the best performing implementation available to us: Thrust [53] version 1.9, MKL [78] version 12.0, CUBLAS [79] version 7.0, CUSPARSE [79, 80] version 7.0, and Rodinia [72] 3.0. In selecting the reference for each benchmark, we chose CUBLAS, CUSPARSE and Thrust for GPUs and MKL for CPUs where possible. Particularly, MKL, CUBLAS, and CUSPARSE come with their own offline tuning and then heuristic version selections for parameterization of different architectures. When using Rodinia, we chose the best known hand-optimized version from the benchmark suite. For example, for the Rodinia CPU references, we pick the best result among the OpenMP version (with icc -O3) and the OpenCL version on top of the Intel and AMD OpenCL CPU stacks.

The evaluated input datasets include a 16M integer array for `Scan`, two 512K-by-128 (tall-and-skinny, TS) and 128-by-512K (short-and-fat, SF) matrices for `SGEMV`, a pair of 4K-by-4K and transposed 4K-by-4K matrices for `DGEMM`, the `bcsstk18` matrix and 100 iteration for `SpMV`, the `kdd_cup` dataset (default in Rodinia) for `KMeans`, and the `graph1MW_6` dataset (default in Rodinia) for `BFS`.

The evaluated codelets for each benchmark are summarized as follows. `Scan` (inclusive) has 4 codelets, and reuses 2 codelets from exclusive scan and 4 from reduction. `SGEMV` has 1 codelet, and reuses 1 from dot-product, 2 from order-preserving reduction, and 3 from non-order-preserving reduction. `DGEMM` has 1 codelet, and reuses 2 from dot-product and 1 from reduction; `SpMV` has 1 codelet, and reuses 1 from sparse scalar multiplication, 1 from

sparse dot-product, and 4 from reduction. `KMeans` has 1 codelet, and reuses 1 from difference, 4 from reduction, 1 from minima selection, and 1 from a gemm-like operation. `BFS` has 1 codelet, and reuses 1 from edge visiting, and 1 from vertex visiting.

For the benchmarks with regular memory access patterns and no data-dependent control flow, such as `Scan`, `SGEMV`, and `DGEMM`, we use offline profiling. For iterative applications, such as `SpMV` and `KMeans`, we apply online profiling using techniques similar to [63] and only profile the first iteration. For irregular but non-iterative applications, such as `BFS`, we use offline profiling on synthetic random graphs.

8.2 Performance Results

Figure 8.1 shows the performance of the six benchmarks on the three evaluation architectures, comparing the TANGRAM implementation to the reference implementation for each. In presenting the results, performance is normalized to the best performing implementation for each benchmark (highest bar), thereby showing the relative performance of the implementations being compared.

8.2.1 Scan

TANGRAM's `Scan` consistently outperforms Thrust's for all devices. TANGRAM's `Scan` is expressed with codelets of different simple scan algorithms, including sequential scan, tree-structure scan, and recursive scan. Particularly, each scan call in a recursive scan codelet can be mapped to different scan codelets to fit the architectural hierarchy and to further enable high performance portability. Given a target architecture, offline profiling can be used to select the best version from a range of competitive ones.

Besides selecting appropriate compositions, coarsening factors, and tiling factors, TANGRAM also benefits from fusing `maps`. `Scan` is commonly written as either a scan-scan-add or a reduce-scan-scan algorithm [81]. In either case, the middle scan lacks parallelism and blocks fusion. However, the three stages can be fused by implementing the middle scan in a streaming or sliding fashion using atomic operations [82, 83]. In TANGRAM, this sliding

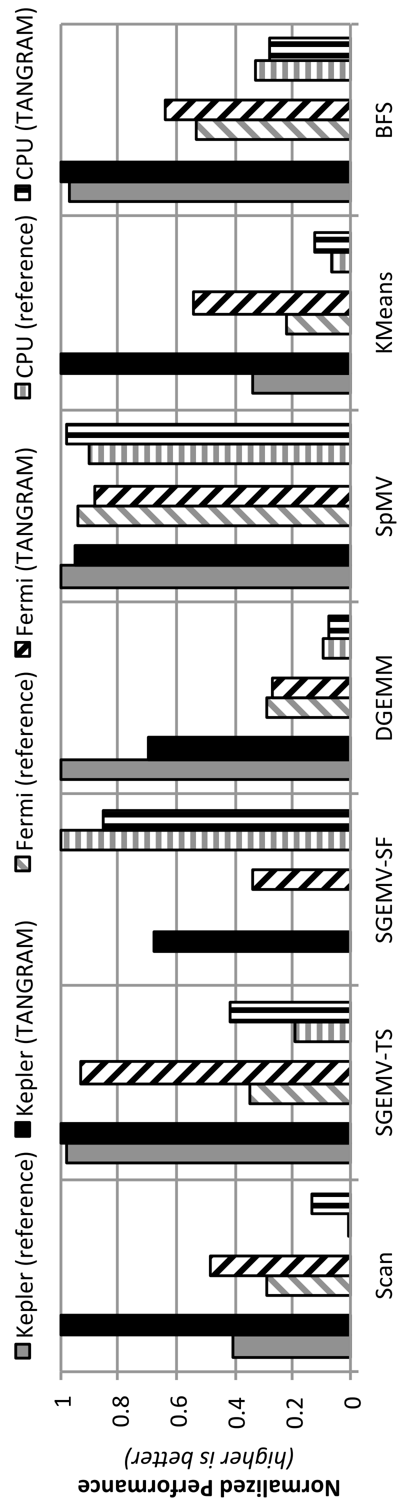


Figure 8.1: TANGRAM Performance Results

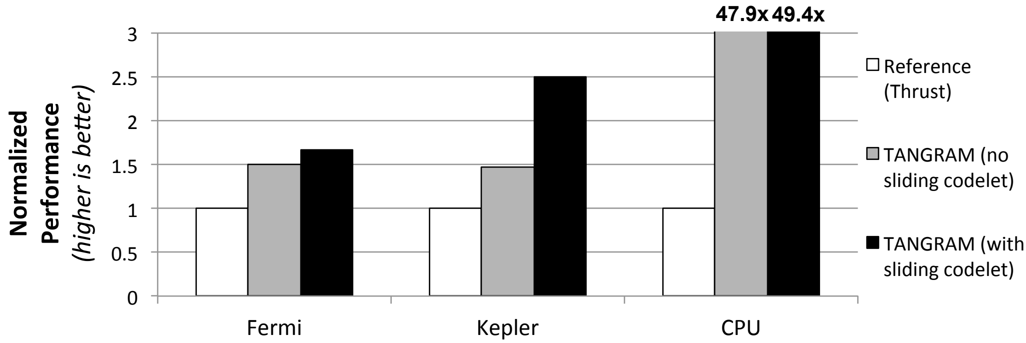


Figure 8.2: `Scan` Results with or without the Sliding Codelet (Normalized to the Corresponding Thrust Results)

scan is implemented as a codelet in the scan spectrum using `map`, enabling TANGRAM to automatically generate a composition that fuses the three stages.

Figure 8.2 compares TANGRAM's scan to the reference with and without the sliding codelet (which enables fusion). Even without the extra codelet, TANGRAM still outperforms Thrust, due to better choice of `partition` parameters, which are labeled as tunable in TANGRAM. The addition of the sliding codelet has more impact on GPUs than CPUs because GPU cachelines have shorter lifetimes than CPU ones, so applying fusion is more critical in order to avoid reloading intermediate data. Note that Thrust's CPU result[1] is 47.9-49.4x slower than TANGRAM's due to an inefficient CPU implementation in Thrust version 1.9.

8.2.2 SGEMV

Parallelism of `SGEMV` highly depends on the height of input matrix. Therefore, a tall-and-skinny (TS) matrix is used as a test case with high parallelism and a short-and-fat (SF) matrix as a test case with low parallelism.

In the TS matrix, TANGRAM's `SGEMV` surprisingly outperforms MKL's on the CPU by a factor of 2.18x while delivering comparable performance to CUBLAS' on Kepler and outperforms CUBLAS' on Fermi by a factor of 2.69x. In this particular evaluation, since MKL's `SGEMV` delivers only less than half of the memory bandwidth, we believe it is mistuned. A similar conclusion

[1]Note that Thrust CPU `Scan` is confirmed as multi-threaded.

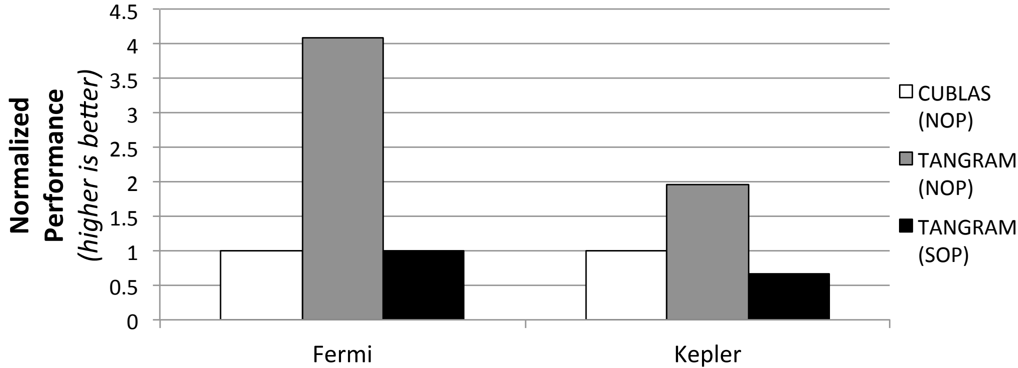


Figure 8.3: SOP and NOP `SGEMV`-SF Results on GPUs (Normalized to the Corresponding CUBLAS Results)

is also applied to Fermi's `SGEMV`. This demonstrates that the current industry practice falls short in keeping performance-critical libraries well tuned for each generation of hardware.

In the SF matrix, we discover[2] CUBLAS does not implement the "standard" `SGEMV` (denoted as SOP, sequential-order-preserving), which preserves the sequential order of the dot-product. A "non-standard" `SGEMV` (denoted as NOP, non-order-preserving) generates different rounding error from the SOP one, and might impact some applications. The SOP `SGEMV` only allows the dot-product following the sequential order, so NOP reduction codelets (like Figure 4.3 (b), (c), and (d) but using float) must be excluded. Two reduction codelets used in the SOP `SGEMV` are sequential reduction codelet (similar to Figure 4.3 (a)) and a sliding fashion [83] sequential reduction codelet. Therefore, the only difference between SOP and NOP is the reduction codelets used in the dot-product. This also shows the productivity of TANGRAM framework without kernel redevelopment.

Figure 8.3 compares the SOP and NOP `SGEMV` with the SF matrix on the GPUs. For NOP `SGEMV`, TANGRAM's code outperforms CUBLAS' on Kepler and Fermi by factors of 1.97x and 4.09x respectively. It is worth mentioning that TANGRAM's SOP `SGEMV` has performance comparable to that of CUBLAS' NOP `SGEMV` on Fermi.

[2]by examining the output rounding errors of specially designed matrices

8.2.3 DGEMM

TANGRAM's `DGEMM` performs within 30% difference of all reference implementations. The presented TANGRAM CPU result is based on generic codelets, though, as mentioned in Section 4.2.1, the TANGRAM framework allows easy integration of intrinsics through __env. The version using AVX intrinsics can gain another 7% performance improvement.

One difficulty in achieving good performance in `DGEMM` is that it is bounded by instruction throughput. The current implementation of TANGRAM relies on the backend C (icc) or CUDA compiler (nvcc) to generate good quality code. Therefore, our TANGRAM Kepler result (717 GFLOPS) achieves 70% performance of CUBLAS (1,027 GFLOPS). In the future, we will likely employ more optimization passes in our compiler and provide a code generation path to PTX or assembly code to better control these low-level factors. For example, one such factor is register bank conflicts [84], which are hard to address at the source-code level.

8.2.4 SpMV

TANGRAM's `SpMV` delivers performance comparable (within 10%) to that of all reference implementations, doing slightly worse than CUSPARSE on the GPUs, and slightly better than MKL on the CPU. In `SpMV`, two candidate kernels are generated for each architecture. Online profiling [63] is applied to the first iteration to select the best version, and the overhead of online profiling is less than 0.8%.

Traditional implementations [85, 86] only consider the warp-centric dot-product and the scalar dot-product. The former tends to have a better memory access pattern but less parallelism than the latter. Compared to the traditional implementations, TANGRAM explores more combinations of compositions with built-in optimizations, such as transposition on GPU scratchpad memory.

Figure 8.4 shows how built-in optimizations impact the final performance. In this evaluation, the result with TANGRAM's optimizations is a version very similar to the scalar dot-product. As mentioned in Section 6.6.2, TANGRAM's composition process prefers higher parallelism by preferring the rules that generate a *distribute*. Although the scalar dot-product might have

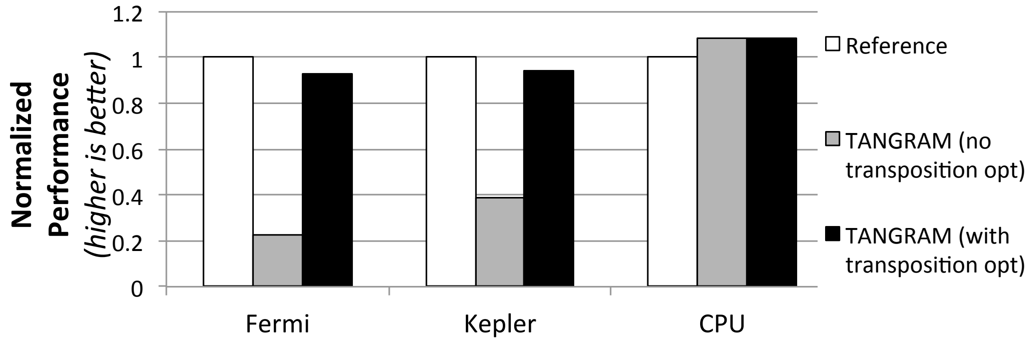


Figure 8.4: `SpMV` Results with and without Transposition Optimization (Normalized to the Corresponding References)

a worse memory access pattern, TANGRAM's optimizers can still improve its performance by applying proper optimizations, such as transposition on GPU scratchpad, or selecting a proper tile size for CPU caches. Particularly, TANGRAM's GPU implementation is similar to [87]. In the end, the results show the built-in optimizations significantly improve performance of `SpMV` in TANGRAM by up to 4.08x on GPUs. Note the CPU results are not sensitive to TANGRAM's optimizers, because CPUs tend to have larger caches to tolerate different tiling sizes.

8.2.5 KMeans

TANGRAM's `KMeans` consistently outperforms Rodinia's for all devices (the best performing CPU version among Rodinia implementations was OpenMP).

For `KMeans`, TANGRAM generates seven candidates (with different coarsening factors and data placements) for GPUs and four candidates for CPUs. The online profiling is applied to the first iteration, and the total overhead is less than 2%.

As mentioned in Section 6.6.2, TANGRAM applies the rules that have favorable locality among those generating a *distribute*, and consequently outperforms Rodinia's, all of which access the loops (the feature loop and the cluster loop) in a suboptimal order.

8.2.6 BFS

TANGRAM's `BFS` performs within 10% difference of all reference implementations, doing slightly better than Rodinia's on the GPU, and slightly worse on the CPU (the best performing CPU version of Rodinia BFS is the OpenCL version on top of Intel's stack). The evaluated `BFS` only includes the same vertex-based algorithm that Rodinia uses for fair comparison.

The vertex status checking (for `g_graph_mask`) and edge index fetching (for `g_graph_nodes`) of `BFS` are parallelizable. While Rodina's parallelizes the former one but serialize the latter one, TANGRAM's parallelizes both, since TANGRAM tends to apply a rule with higher parallelism.

Figure 8.5 shows the execution breakdown of each iteration for both GPU versions. TANGRAM's version tends to perform better than Rodinia's when the number of active vertices (which is proportional to workload size) is large. However, due to a very low number of edges per vertex in some iterations, like the 11th iteration, TANGRAM's versions perform slightly worse than Rodinia's.

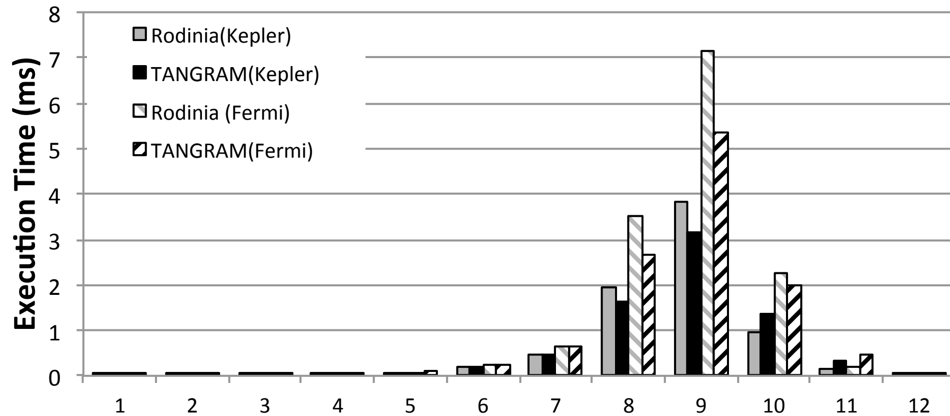


Figure 8.5: `BFS` GPU Performance Breakdown

8.3 Discussion

TANGRAM language enables expression for interchangeable codelets, allowing recursive calls to adapt different architectural hierarchies and tunable qualifiers for parameterization tuning to adapt resource sizes. Therefore,

the TANGRAM compiler potentially can choose alternative algorithms or optimizations for a particular computation to achieve better performance.

Algorithms. In our evaluation, TANGRAM's `DGEMM`, `KMeans`, and `BFS` use the same algorithms as the references. TANGRAM's `SGEMV` uses the same algorithm as the standard BLAS and MKL, while CUBLAS uses a different algorithm for the SF matrix. For `Scan` and `SpMV`, we cannot confirm whether the algorithms are the same, since the references are close-sourced. However, particularly for `Scan`, we believe TANGRAM synthesized a different combination of scan algorithms compared to Thrust.

Synthesized Kernels. Most differences among the synthesized kernels for different types of architectures (for example, CPUs and GPUs) are either algorithmic or loop structure due to different hierarchies. For the same type of architectures (for example, Fermi and Kepler GPUs), the differences mainly come from different parameters or data placement. The only exception happens in `Scan`: its algorithmic combination changes from Fermi to Kepler, due to the high efficiency of Kepler's shuffle instructions.

Reasons for High Performance. We summarize the major reasons why TANGRAM can achieve performance better than or comparable to the references. TANGRAM potentially can deliver better algorithmic combination to match the architectural hierarchy (`Scan`), more parallelism (`SGEMV`, `SpMV` and `BFS`), better locality (`Kmeans`), better parameters (`Scan`) or better data placement (`Kmeans` and `SpMV`). For `DGEMM`, TANGRAM did not outperform MKL or CUBLAS, since the references are written in assembly.

8.4 Performance Comparison to Existng Composition-based Language

As discussed in Section 4.5, languages [40, 59, 55] with composition rules can potentially provide functionality of adaptation to architectural hierarchy similar to what TANGRAM does. Petabricks [55] is the most similar work to TANGRAM, allowing the user to define **codelet**-like functions (called *transforms* and *rules*), supporting **composition** and parameter tuning, and trying to achieve performance portability on CPUs and GPUs [88].

The major difference between TANGRAM and Petabricks is **architectural optimization**. TANGRAM introduces architectural hierarchy mod-

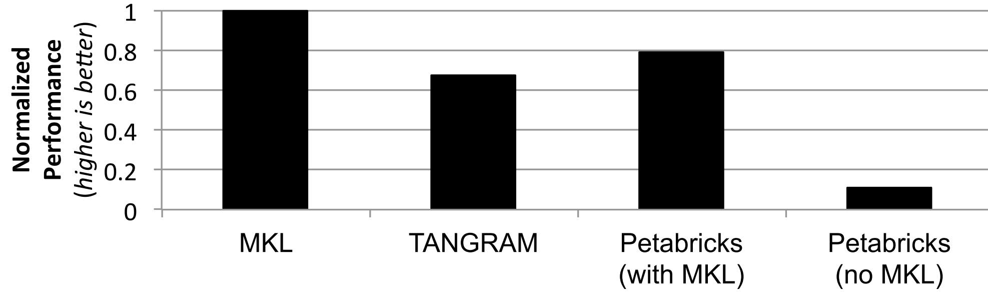


Figure 8.6: Comparison between TANGRAM and Petabricks using `DGEMM` on CPU

els and corresponding rules to guide composition and optimization processes, and focuses on architectural optimizations themselves. Compared to TANGRAM, Petabricks directly relies on autotuning (using evolutionary algorithms, particularly) for design space search, and focuses on task scheduling, and selection of proper algorithms or libraries for particular input data. Lack of architectural hierarchy models could obstruct possible composition and potential architectural optimization for the target architecture, then prevent exploration of certain versions, and possibly lead to a suboptimal result. Meanwhile, lack of general architectural optimizations could cause catastrophic performance degradation for generated code.

To demonstrate this difference, a common benchmark, `DGEMM`, is evaluated. Figure 8.6 shows Petabricks can achieve 79% and 11% of MKL performance, with and without calling MKL `DGEMM` internally[3] respectively, while TANGRAM can achieve 70% of MKL performance (without calling MKL `DGEMM` internally). These results imply that Petabricks highly relies on high-performance base *rules* (atomic codelets in TANGRAM). We also observe that the released package of Petabricks did not optimize function inlining, thread spawning, and branch divergences of version selection, so it achieved only 79% of MKL performance even with internal MKL `DGEMM` calls. This evaluation demonstrates that architectural optimizations are crucial to achieve high performance. Other important differences include TANGRAM's support for cooperative codelets, which is crucial for better utilization of SIMD execution on modern architectures. TANGRAM also introduces static pruning in composition to select competitive candidates before profiling.

[3]The `DGEMM` in Petabricks calls MKL by default and disables all other rules. For fair comparison, we re-enable all of the rules and optionally enable MKL `DGEMM`.

CHAPTER 9

CONCLUSION AND FUTURE WORK

9.1 Conclusion

In this dissertation, I have discussed existing techniques tackling performance portability challenges that result from major hardware differences, and surveyed current programming systems attempting to resolve these challenges. I have further identified the key issues of current programming systems for achieving performance portability, classified them in two categories, language and compiler, and investigated them in terms of the design space.

I present a programming system, TANGRAM, addressing these two major kinds of issues, by presenting a new programming language with its compiler. The language is designed for effectively expressing a design space with high coverage by enabling users to define compute patterns via atomic and compound codelets and built-in containers and primitives. TANGRAM is a modular, composition-based programming language that can deliver effective expression for a design space as well as productivity and maintainability.

On the other hand, the compiler is designed to effectively explore the design space and facilitate many high-level optimization techniques. I have developed a user-friendly compiler infrastructure that is specialized for compositions and design space exploration. I have further exploited the infrastructure to design the compiler that incorporates a series of static kernel synthesis and selection techniques such as hierarchical composition, coarsening, data placement, and autotuning. The TANGRAM programming system supports performance portability through compositions of user-defined architecture-neutral code into high-performance kernels customized for different architectural hierarchies.

This dissertation provides comprehensive description of the architectural hierarchy model, the composition mechanism, and the code generation. The

results show that TANGRAM can achieve performance of at least 70%, and in some cases multiple times, of various well-known reference implementations on various architectures.

9.2 Future Work

As for future work, I believe the TANGRAM programming system can be further improved along the following major directions:

- Extending the language and compiler to support more computation patterns in real-world applications
- Extending the language and compiler to support complex tuning constraints and the precision control
- Merging the language interface with C++ or other standards
- Developing more compiler passes for both analyses and optimizations
- Redesigning the current composition planner to support more sophisticated design space exploration
- Introducing a new type of memory level in hardware abstraction
- Extending the compiler to support FPGAs, Integrated system, and/or clusters

The current language is implemented as a limited extension of C++, and the compiler is designed to focus at the kernel synthesis. Therefore, the current language might not always fit every existing software, and the current compiler might have a trouble handling tasks like I/O or memory transfer, which might be important in certain applications. To achieve those goals, we need to extend the language and the compiler.

TANGRAM currently provides tuning knobs through `__tunable` qualifiers. In the current implementation, these tuning knobs are parameterized during code generation, and then specified later. However, in practice, programmers might be able to provide information, such as a reasonable range or legitimate values, for these parameters. Therefore, it is useful to provide this kind of interface in the language level. Also, data precision transformations are widely

applicable for energy saving or performance boosting. TANGRAM currently does not include optimizations related to precision-related transformations. Programmers currently can only encode precision-related transformations using different codelets to enable these kinds of optimizations. It is useful to provide native support for precision-related transformations by introducing keywords in the language level, and related optimizations in the compiler level.

TANGRAM is a new programming language, and programmers generally resist adopting new languages. However, TANGRAM in fact shares a significant amount of language syntax with the C++17 standard, so it can potentially be merged with C++17. Also, if we provide parsers for existing composition-based languages, Petabricks, Sequoia, and even Halide, theoretically the TANGRAM compiler can be exploited to support compositions in those languages.

The current compiler implementation still has a limited number of analyses and optimizations. In order to achieve high performance across devices for general applications, a huge number of compiler analyses and optimizations are required. The TANGRAM compiler infrastructure should be able to facilitate compiler development.

The current composition mechanism is driven by a heuristic that is based on my personal understanding of high-performance parallel programming on CPUs and GPUs. It might not be general or practical for devices that I never studied or future devices. Therefore, we might need to develop a new composition algorithm when the current one is not beneficial, or study a self-guided composition algorithm through machine learning or a reconfigurable algorithm through micro-benchmarking.

The current TANGRAM system does not support memory abstraction for hardware. Therefore, it cannot easily describe characteristics of near-memory processors, and guide design space exploration for them. In order to support these energy-efficient near-memory processors, we need to extend hardware abstraction to model memory.

Support for FPGAs, integrated systems and clusters might become very challenging, since FPGAs have feasible hierarchies and integrated systems and clusters are highly sensitive to communication. It might require more sophisticated hardware abstraction (e.g. introducing new abstraction for memory and/or communication), composition rules (e.g new composition rules to

split workloads for heterogeneous processors), composition algorithms (e.g machine-learning-based or model-based composition algorithms for FPGA due to no fixed hierarchy) and, of course, optimizations. Also, code generation requires a certain degree of modification to support efficient backends for those devices. Hopefully, the TANGRAM compiler infrastructure should be general enough to facilitate modification for future work.

REFERENCES

[1] Khronos group, "The OpenCL specification," *Version 2.0*, 2015.

[2] K. Gregory and A. Miller, *C++ AMP: Accelerated Massive Parallelism with Microsoft® Visual C++®*. O'Reilly Media, Inc., 2012.

[3] "OpenMP application programming interface," *Version 4.5*, 2015.

[4] "OpenACC," http://www.openacc.org/.

[5] "OpenCL Zone." [Online]. Available: http://developer.amd.com/tools-and-sdks/opencl-zone/

[6] "Beignet." [Online]. Available: https://01.org/beignet

[7] "NVIDIA OpenCL SDK." [Online]. Available: https://developer.nvidia.com/opencl

[8] "Mali OpenCL SDK." [Online]. Available: http://malideveloper.arm.com/develop-for-mali/sdks/mali-opencl-sdk/

[9] "PowerVR SDK." [Online]. Available: http://community.imgtec.com/developers/powervr/

[10] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng, "Twin Peaks: A Software Platform for Heterogeneous Computing on General-purpose and Graphics Processors," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, 2010, pp. 205–216.

[11] "Intel SDK for OpenCL Applications." [Online]. Available: https://software.intel.com/en-us/intel-opencl

[12] "OpenCL Development Kit for Linux on Power." [Online]. Available: http://www.alphaworks.ibm.com/tech/opencl

[13] "OpenCL Development Kit for Linux on Power." [Online]. Available: http://www.alphaworks.ibm.com/tech/opencl

[14] P. Jääskeläinen, C. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "pocl: A performance-portable OpenCL implementation," *International Journal of Parallel Programming*, pp. 1–34, 2014.

[15] "FreeOCL." [Online]. Available: https://code.google.com/p/freeocl/

[16] H.-S. Kim, I. El Hajj, J. Stratton, S. Lumetta, and W.-M. Hwu, "Locality-centric thread scheduling for bulk-synchronous programming models on CPU architectures," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2015, pp. 257–268.

[17] "MAGMA." [Online]. Available: http://icl.cs.utk.edu/magma/software/

[18] "OpenCV." [Online]. Available: http://opencv.org/

[19] H.-S. Kim, "Performance portable compiler techniques for bulk-synchronous programming models on CPU architectures," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2015.

[20] J. A. Stratton, C. Rodrigues, I. Sung, N. Obeid, L. Chang, N. Anssari, G. D. Liu, and W. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *IMPACT Technical Report*, 2012.

[21] J. Stratton, N. Anssari, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, G. D. Liu, and W.-m. Hwu, "Optimization and architecture effects on GPU computing workload performance," in *Innovative Parallel Computing (InPar), 2012*, 2012, pp. 1–10.

[22] J. Stratton, "Performance portability of parallel kernels on shared-memory systems," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2013.

[23] A. Magni, C. Dubach, and M. F. O'Boyle, "A large-scale cross-architecture evaluation of thread-coarsening," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, p. 11.

[24] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters," in *Proceedings of the 26th ACM International Conference on Supercomputing*, 2012, pp. 341–352.

[25] J. A. Stratton, S. S. Stone, and W. W. Hwu, "MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs," in *Languages and Compilers for Parallel Computing*, J. N. Amaral, Ed., 2008, pp. 16–30.

[26] N. Rotem, "Intel OpenCL Implicit Vectorization Module," 2011.

[27] R. Karrenberg and S. Hack, "Improving Performance of OpenCL on CPUs," in *Proceedings of the 21st International Conference on Compiler Construction*, 2012, pp. 1–20.

[28] S. Muralidharan, M. Garland, B. Catanzaro, A. Sidelnik, and M. Hall, "A collection-oriented programming model for performance portability," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2015, pp. 263–264.

[29] S. Muralidharan, M. Garland, A. Sidelnik, and M. Hall, "Designing a tunable nested data-parallel programming system," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 4, pp. 47:1–47:24, Dec. 2016. [Online]. Available: http://doi.acm.org/10.1145/3012011

[30] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng, "Twin Peaks: A Software Platform for Heterogeneous Computing on General-purpose and Graphics Processors," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, 2010, pp. 205–216.

[31] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu, "Adaptive cache management for energy-efficient gpu computing," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.11 pp. 343–355.

[32] X. Chen, S. Wu, L.-W. Chang, W.-S. Huang, C. Pearson, Z. Wang, and W.-M. W. Hwu, "Adaptive cache bypass and insertion for many-core accelerators," in *Proceedings of International Workshop on Manycore Embedded Systems*, ser. MES '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2613908.2613909 pp. 1:1–1:8.

[33] NVIDIA, "Tuning CUDA Applications for Kepler." [Online]. Available: http://docs.nvidia.com/cuda/kepler-tuning-guide/

[34] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *High Performance Computing, Networking, Storage and Analysis, International Conference for*, 2008, pp. 1–11.

[35] J. Kurzak, S. Tomov, and J. Dongarra, "Autotuning GEMM kernels for the Fermi GPU," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, no. 11, pp. 2045–2057, 2012.

[36] G. Chen, B. Wu, D. Li, and X. Shen, "PORPLE: An extensible optimizer for portable data placement on GPU," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, 2014, pp. 88–100.

[37] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting memory access patterns to improve memory performance in data-parallel architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 105–118, 2011.

[38] NVIDIA, "Tuning CUDA Applications for Maxwell." [Online]. Available: http://docs.nvidia.com/cuda/maxwell-tuning-guide/

[39] NVIDIA, "CUDA Pro Tip: Optimized Filtering with Warp-Aggregated Atomics," http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/.

[40] G. E. Blelloch, "NESL: A nested data-parallel language," Pittsburgh, PA, USA, Tech. Rep., 1992.

[41] S. Jones, "Introduction to dynamic parallelism," in *GPU Technology Conference Presentation*, 2012.

[42] W.-m. W. Hwu, *Heterogeneous System Architecture: A New Compute Platform Infrastructure*. Morgan Kaufmann, 2015.

[43] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on GPUs," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 528–540.

[44] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "LaPerm: Locality aware scheduler for dynamic parallelism on gpus," in *The 43rd International Symposium on Computer Architecture (ISCA)*, June 2016.

[45] M. S. Orr, B. M. Beckmann, S. K. Reinhardt, and D. A. Wood, "Fine-grain task aggregation and coordination on GPUs," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, 2014, pp. 181–192.

[46] I. El Hajj, J. Gómez-Luna, C. Li, L.-W. Chang, D. Milojicic, and W.-m. Hwu, "KLAP: Kernel launch aggregation and promotion for optimizing dynamic parallelism," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, 2016, pp. 1–12.

[47] Y. Yang and H. Zhou, "CUDA-NP: Realizing nested thread-level parallelism in GPGPU applications," in *ACM SIGPLAN Notices*, vol. 49, no. 8, 2014, pp. 93–106.

[48] G. Chen and X. Shen, "Free launch: optimizing GPU dynamic kernel launches through thread reuse," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 407–419.

[49] D. Merrill, M. Garland, and A. Grimshaw, "Policy-based tuning for performance portability and library co-optimization," in *Innovative Parallel Computing (InPar), 2012*, 2012, pp. 1–10.

[50] M. Püschel, J. M. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson, "Spiral: A generator for platform-adapted libraries of signal processing alogorithms," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 21–45, 2004.

[51] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1, pp. 3–35, 2001.

[52] Intel Corporation, "C/C++ Extensions for Array Notations Programming Model," http://software.intel.com/en-us/node/522649.

[53] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for CUDA," *GPU Computing Gems Jade Edition*, p. 359, 2011.

[54] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach, "Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, 2015, pp. 205–217.

[55] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "PetaBricks: A language and compiler for algorithmic choice," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 38–49.

[56] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU concurrency with elastic kernels," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013, pp. 407–418.

[57] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu, "Zorua: A holistic approach to resource virtualization in GPUs," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–14.

[58] Khronos group, "SYCL Provisional Specification," *Version 2.2*, 2016.

[59] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[60] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[61] J. Hoberock, "The Parallelism TS Should be Standardized," http://www.open-std.org/Jtc1/sc22/wg21/docs/papers/2016/p0024r1.

[62] H. Guihot, "Renderscript," in *Pro Android Apps Performance Optimization*. Springer, 2012, pp. 231–263.

[63] L.-W. Chang, H.-S. Kim, and W.-m. Hwu, "DySel: Lightweight dynamic selection for kernelbased data-parallel programming model," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2016, pp. 667–680.

[64] J.-F. Dollinger and V. Loechner, "Adaptive runtime selection for GPU," in *42nd International Conference on Parallel Processing*, 2013, pp. 70–79.

[65] L. Li, U. Dastgeer, and C. Kessler, "Adaptive off-line tuning for optimized composition of components for heterogeneous many-core systems," in *International Conference on High Performance Computing for Computational Science*, 2012, pp. 329–345.

[66] J. Srinivas, W. Ding, and M. Kandemir, "Reactive tiling," in *Code Generation and Optimization, 2015 IEEE/ACM International Symposium on*, 2015, pp. 91–102.

[67] J. R. Wernsing and G. Stitt, "Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing," in *ACM SIGPLAN Notices*, vol. 45, no. 4, 2010, pp. 115–124.

[68] M. J. Voss and R. Eigemann, "High-level adaptive program optimization with ADAPT," in *ACM SIGPLAN Notices*, vol. 36, no. 7, 2001, pp. 93–102.

[69] L.-W. Chang, I. El Hajj, C. Rodrigues, J. Gómez-Luna, and W.-m. Hwu, "Efficient kernel synthesis for performance portable programming," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, 2016, pp. 1–13.

[70] K. Goto and R. A. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 3, p. 12, 2008.

[71] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *ACM SIGPLAN Notices*, vol. 46, no. 8, 2011, pp. 267–276.

[72] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, 2009, pp. 44–54.

[73] Intel, "Vectorizer knobs," https://software.intel.com/en-us/node/540483.

[74] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, International Symposium on*, 2004, pp. 75–86.

[75] D. Quinlan, "ROSE: Compiler support for object-oriented frameworks," *Parallel Processing Letters*, vol. 10, no. 02n03, pp. 215–226, 2000.

[76] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A source-to-source compiler infrastructure for multicores," *Computer*, vol. 42, no. 12, 2009.

[77] "The Matrix Market," http://math.nist.gov/MatrixMarket/.

[78] "Intel Math Kernel Library," http://software.intel.com/en-us/articles/intel-mkl/.

[79] NVIDIA, *CUBLAS Library User Guide*, v7.0 ed., 2015.

[80] NVIDIA, *CUDA CUSPARSE Library*, v7.0 ed., 2015.

[81] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli, "Fast scan algorithms on graphics processors," in *Proceedings of the 22Nd Annual International Conference on Supercomputing*, 2008, pp. 205–213.

[82] S. Yan, G. Long, and Y. Zhang, "StreamScan: Fast scan algorithms for GPUs without global barrier synchronization," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013, pp. 229–238.

[83] J. Gomez Luna, L.-W. Chang, I.-J. Sung, N. Guil Mata, and W.-M. W. Hwu, "In-place data sliding algorithms for many-core architectures," in *Parallel Processing, International Conference on*, 2015 (in press).

[84] J. Lai and A. Seznec, "Performance upper bound analysis and optimization of sgemm on Fermi and Kepler GPUs," in *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, 2013, pp. 1–10.

[85] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Corporation, NVIDIA Technical Report NVR-2008-004, 2008.

[86] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010, pp. 63–74.

[87] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 769–780.

[88] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe, "Portable performance on heterogeneous architectures," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. 48, no. 4, 2013, pp. 431–444.