AUTOTUNING, CODE GENERATION AND OPTIMIZING COMPILER TECHNOLOGY FOR GPUS

by

Malik Muhammad Zaki Murtaza Khan

A Dissertation Presented to the
FACULTY OF THE USC GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

May 2012

I dedicate this thesis to my loving mother, Mrs. Tanveer Fatima Malik (Late)and to my father Mr. Malik Shahin Murtaza Khan.

Acknowledgements

I would like to thank The Almighty, for all the blessings throughout my life. I would thank my parents for their unconditional support in every endeavour of my life. A special thanks is to my advisor, Dr. Mary Hall for her guidance and directions in my PhD research. I am fortunate to have Dr. Aiichiro Nakano and Dr. Victor Prasanna as dissertation defence committee members. A special thanks to Dr. Chun Chen, Dr. Jacqueline Chame, Gabe Rudy and Protonu Basu for many fruitful discussions.

I must acknowledge and thank my wife, Wajiha Asad Malik, my son, Muhammad Hadi Abbas, for all the support and wonderful moments we share every day. I also must acknowledge the support of my brothers Muhammad Taqi Murtaza, Muhammad Naqi Murtaza and Muhammad Razi Murtaza. Thank you.

This research was initially funded in part by MS-PhD scholarship(NUST, Pakistan). Later it was funded in part by DARPA contract HR0011-10-9-0008, Department of Energy Office of Science Award DE-FC02-06ER25765, and National Science Foundation award CCF-1018881.

Table of Contents

List of Figures

List of Tables

Abstract

Graphics Processing Units (GPUs) have evolved to devices with teraflop-level performance potential. Application developers have a tedious task in developing GPU software by correctly identifying parallel computation and optimizing placement of data for the parallel processors in such architectures. Further, code optimized for one architecture may not perform well on different generations of even the same processor family. Many manually tuned GPU solutions would need a complete rewrite on a different architecture, hence more programmer time and effort. High-performance computing on GPUs can be facilitated by programming models and programming frameworks that attempt to reduce the amount of time and effort needed to develop GPU applications.

This thesis describes a compiler framework for automatically generating and optimizing parallel code for GPUs (CUDA code for Nvidia GPUs), relieving programmers of the tedious work of parallelizing sequential code. The framework describes a script-based compiler for CUDA code generation combining:

- a Transformation Strategy Generator (TSG), which automatically generates multiple scripts representing different optimization strategies;
- an Autotuning System that automatically generates a set of code variants and selects the best among them through empirical evaluation.

An underlying loop transformation and code generation framework takes TSG-generated scripts as input and generates CUDA code; thus enabling an end-to-end system to produce high performance solutions for scientific computations on a given GPU architecture.

This flexible organization enables the system to explore a large optimization search space, simultaneously targeting different architecture features, but constrained by data dependences and guided by data reuse and the best heuristics from manual tuning. The system tailors the generated code to yield high-performance results for different GPU generations, data types and data sets.

The key contributions of this thesis include:

(1) the meta-optimizer, TSG,

(2) a search and autotuning mechanism,

(3) integration with a script-based compiler framework, resulting in an end-to-end automatic parallelization system.

(4) performance-portable code generation for the Nvidia GTX-280 and Nvidia Tesla C2050 Fermi architectures, and

(5) performance gains of up to 1.84x over linear algebra kernels in the manually-tuned Nvidia CUBLAS library, and up to 2.03x for a set of scientific, multimedia and imaging kernels over a state-of-the-art GPU compiler.

Chapter 1

Introduction

Peak performance of graphics processors (GPUs) is now in the Teraflop range for a single device, and general purpose code on GPUs has demonstrated up to two orders of magnitude performance gains as compared to conventional CPUs [VD08a, Wol08, DMV^{+}08, BG09]. In spite of this enormous potential, it is very difficult to develop GPU applications that make efficient use of all the architectural features and achieve high performance, particularly given the significant architectural changes across GPU generations. For example, consider Nvidia GPUs.

- Nvidia GPU chips are partitioned into multiple streaming multiprocessors (SMs), each of which have multiple cores. Details of how a computation gets mapped to these SMs affects the performance enormously.

- Explicit data movement from the CPU address space to the GPU memory hierarchy is needed.

- The GPU memory hierarchy must be managed explicitly in software to hide memory latency.

- Improving memory bandwidth must also be managed explicitly using memory operations supported in software, and dedicated hardware mechanisms.

- Maximum performance on a GPU may depend on tuning multiple parameters in the application.

Thus the programmer must explicitly manage available parallelism and the heterogeneous memory hierarchy. Failing to address any of these aspects appropriately may change the mapping of the computation onto the physical resources and thus may severely affect performance. Tools to simplify the programming and performance tuning process have the potential to increase the accessibility of this important technology and improve performance of the resulting code.

1.1 State-of-the-art in GPU Programming Tools

Nvidia has introduced their parallel programming extension called CUDA (Compute Unified Device Architecture), which has demonstrated high performance for many scientific applications [Nvi08a, Wol08]. The CUDA parallel programming model tries to provide programmers with a modest set of language extensions to exploit the parallelism and effectively use the memory hierarchy of the Nvidia GPU devices [Nvi08a]. As previously stated, multiple aspects of programming Nvidia GPUs must be explicitly managed to maximize the performance gain. Researchers have reported days of effort to program high performing solutions for well understood problems like matrix-matrix multiplication [Wol08]. In addition to that, the portability of solutions on devices from different vendors or different generation of devices from the same vendor is also a challenging issue.

Contemporary GPU compiler solutions have tried to reduce the complexities of writing low level CUDA code. *hi*CUDA [HA09] and OpenMPC [LME09] are pragma-based compilers which map the parallelism expressed by the application developer to CUDA programs. These approaches are limited in what can be expressed, and may still require significant work by the programmer to partially optimize code. Compilers that *optimize CUDA code* [ULBmWH08, YXKZ10] can be quite effective, but require that the initial CUDA kernel be properly parallelized in a way that will facilitate optimizations.

In contrast, this thesis describes *automatic parallelization*, from sequential C code, typically with subscripts restricted to an affine domain. When applicable, automatic parallelization enables programmers with no GPU programming expertise to use GPUs.

Compared to prior compiler research on automatic parallelization for GPUs, which use a fixed optimization strategy [BBK+08b, BRS10, LVM+10], our autotuning compiler can generate multiple code variants that can be evaluated empirically. We use compiler concepts of dependence and reuse to guide transformations, rather than trying a fixed collection of optimizations that may not be suitable for a computation [CWX+11].

1.2 Research Overview

In this thesis, we focus on developing a set of tools, which work in tandem to come up with high performance solutions for loop nest computations, forming a system which

1. Automatically analyzes a sequential code, expressed in C,
2. Automatically identifies the optimization opportunities in the computation,
3. Proposes multiple optimization strategies, and
4. Employs autotuning to explore a potentially large space of solutions and selects the best implementation.

The first three goals, as mentioned above, rely on extensive compiler analysis of the sequential code and strategic decisions about the optimization strategies based on architecture properties such as capacity limitations of faster memories. Compiler abstractions such as data dependences, data reuse patterns and potential constraints of the hardware and software model of the GPUs help to come up with potential computation decompositions and data placement strategies. There are two major parts of our system, focused on achieving the first three goals:

(1) a Transformation Strategy Generator (TSG), which automatically generates multiple optimization strategies, and is the main focus of this thesis;

(2) an underlying polyhedral loop transformation and code generation framework called CUDA-CHiLL, which takes TSG generated strategies as input and produces CUDA code;

TSG is the most important product of this research work. TSG is able to explore a large optimization space, simultaneously targeting different architecture features, but constrained by data dependences and guided by data reuse and the best heuristics from manual tuning. Additionally, this thesis describes the system which uses TSG capability with an underlying framework for composing high level loop transformations and CUDA code generation.

The complexity of mapping the computation to modern uniprocessor and multi-core architectures has been managed by *autotuning*, that employs empirical techniques to evaluate a set of alternative mappings of computation kernels to an architecture and select the mapping that obtains the best performance. The third part of the system, that addresses the last goal is:

(3) *autotuning mechanism*, which describes employing empirical techniques to evaluate multiple strategies and derive optimization parameter values, exploring a search space of possible implementations.

This thesis introduces a two level autotuning process which is based on separate search strategies at each level to keep the search efficient. Our system first searches for the best combination of computation decomposition and data placement choices. In the second stage we consider load balancing and resource utilization.

This thesis describes an end-to-end system producing optimized solutions for different computation kernels on two different generation GPUs. The results presented in the thesis demonstrate that the overall compiler approach is powerful, yielding performance results comparable to and sometimes exceeding the manually and automatically generated code.

1.3 Example: Matrix-Matrix Multiplication

Consider the example of matrix-matrix multiplication in Figure 1.1, an important kernel used in many scientific computations. The computation is a simple triply-nested

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      c[j][i] = c[j][i] + a[k][i] * b[j][k];
```

Figure 1.1: Matrix-matrix multiplication source code (BLAS)

loop with one computation statement. Many researchers [Che07, BBK+08a, BRS10, YXKZ10] have shown that this simple computation performs well below peak if generated automatically by a compiler. For this reason application developers often prefer to use manually-tuned BLAS libraries for matrix-matrix multiplication to speed up their application.

1.3.1 Limitations of Manually-Tuned Libraries

Manually tuned libraries provide high performance routines for a set of widely used computations, which speed up parts of the application, improving the overall performance of the applications.

There are some limitations with such libraries.

- Many problems require dedicated and exclusive programming effort to devise solutions for GPUs. This approach cannot be extended to cater to the needs of the scientific community to ***generate general application code.*** Every different application needs the same amount of time and effort on the part of developers to achieve a high performance solution.

- There are only so many different solution strategies that can be explored if done manually. There is a good chance that for a certain problem, the programmer may only explore a few possible solutions in a potentially large search space of possibilities, as ***it is too hard*** to evaluate all of them. Thus, developers tend to settle for local maxima, ignoring the large picture due to inability to look through a large search space.

- Different versions of these libraries are needed for different platforms and may need to be modified with every new generation of a specific architecture. Thus ***portability*** is a major concern in developing GPU specific applications.

1.3.2 Matrix-Matrix Multiplication on a GPU

We continue with the example of matrix-matrix multiplication and in light of the major goals of our system. First of all, our system analyzes the triply-nested loop computation (Figure 1.1) and comes up with a computation partitioning strategy. The system follows the computation partitioning analysis with analysis regarding the efficient use of the memory hierarchy. Multiple parallelism and data staging options are pruned out due to these analyzes, leaving a small set to be searched for a high performance solution.

1.3.3 Computation Partitioning

The sequential code can be mapped to the parallel architecture in a variety of ways. There are two main factors which define the computation partitioning on a GPU.

Computation Decomposition is the logical representation of the sub-computations such as 1D, 2D etc layouts.

Computation Partitioning Parameters are the variables which may decide the size of the sub-computations and the number of blocks and number of threads per block.

A minor variation in the computation partitioning parameters or decomposition can lead to significant performance changes. Thus, achieving high performance is partly dependent on efficiently mapping the sub-computations on the available resources like the number of parallel blocks that compute parts of computation, and the number of threads per block doing the amount of work per thread and sub-computations independently. The number of threads per block dictates the number of blocks there are and thus the sub-computations to run concurrently. A minimum number of threads is necessary to hide the latencies associated with the execution unit. An appropriate number of blocks is also necessary to balance the workload on a given set of resources.

1.3.4 Data Placement

In the second stage, we decide where in the memory hierarchy each data structure will be stored. In Nvidia GPUs, the memory hierarchy consists of registers, shared memory, global memory; designated portions of global memory may have different usage and caching behavior.

> **Data Placement Decisions** are made by the system after it analyzes the reuse of data in the computation for possible opportunities of using faster memory in the hierarchy [AK02]. Data reuse patterns are analyzed through the access expressions of the arrays in the computation.
>
> For data reused across threads in a block, the data structure is identified as a candidate for shared memory. Data with an access expression that indicates reuse within a thread are selected for local storage i.e. registers. In our case study of GEMM, the input arrays are identified for shared memory, and the result array uses a register.
>
> **Data Partitioning Parameters** are used to control the data footprint in the target memory, limited by the size of target storage.

The size, layout and accessing mechanism of the data from the global memory to faster but smaller memories, is crucial in achieving high performance. Some problems require the data to be read in a completely different way than they are going to be eventually accessed by the threads. Such subtleties, if ignored may lead to a crucial drop in performance. Equally important is to make decisions about the footprint of the data structures in a specific level of memory hierarchy; sometimes multiple data structures share a portion of the memory hierarchy. The overall footprint is dictated by the choices made with the data partitioning parameters.

1.3.5 Optimization Parameter Combinations

In matrix-matrix multiplication, there are three optimization parameters which define the computation partitioning and data placement. There are multiple possible values that can be bound to these variables. The important considerations, while choosing a particular value for a computation partitioning parameter, are load balancing, parallelism granularity, hiding memory latency and efficient use of registers for thread-specific data. In term of CUDA execution model, these concepts are controlled by to number of parallel blocks handling sub-computations and number of threads per block. The key trade-off here is between the number of threads in a block which can become large enough to limit the number of blocks mapped to an execution unit and can be set to small enough number, where it fails to hide the latencies. Thus our system evaluates all the different valid computation partitioning options both in terms of the computation partitioning parameter values and the layout of the computation. In the case of matrix multiplication, multiple parameter values are tested and the layout of the distribution of the computation is set as 2D, given the 2-dimensional layout of the data structures involved.

Similarly, the data partitioning parameters balance maximizing reuse, minimizing overhead and staying within capacity constraints. Data partitioning is also a tiling problem, like computation partitioning. Tiling commands control the data footprint in faster memory, through tile parameters. Efficient data footprint management through tiling parameter values is coupled with the computation management parameters to obtain high performance code.

1.3.6 Summary

The discussion above has focused on the problem of generating high performance code, for sequential computations, without spending huge time or human effort. We have addressed this problem by presenting a system which transforms sequential code to parallel code by deciding on a set of optimizations leading to code variants and searching a large set of different code variants to identify the best performing code. We believe the uniquely flexible organization of our compiler provides an advantage that yields performance comparable to manually-tuned code, and this advantage is particularly on display when optimizing in the presence of the idiosyncrasies of different GPU architecture generations.

1.4 Contributions

The thesis makes several key contributions, listed as follows:

- *Automatic Code Transformation:*A meta-optimzer called the Transformation Strategy Generator (TSG),automatically generates a collection of code transformation strategies that derive the best-performing implementations of a computation.
- *Autotuning:* It describes an efficient search and autotuning mechanism to derive a single high performing solution from a large search space of possible candidates.
- *Integration with Script-Based Compiler Framework:* It demonstrates an end-to-end system, developing interfaces for integration of the TSG with an underlying script-based compiler framework for code generation.
- *Performance Portability:* It demonstrates an approach that selects customized implementations depending upon problem size or data type (i.e., single vs. double precision) for two Nvidia architectures (GTX-280 and Tesla C2050 Fermi).
- *Experimental Evaluation:* It presents automatically generated code which achieves performance of up to 1.84x over linear algebra kernels in manually-tuned Nvidia CUBLAS library; and up to 2.03x for set of scientific, multimedia and imaging kernels over a state-of-the-art GPU compiler [BUH+08, YXKZ10].

1.5 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 gives a background on GPU and CUDA. Chapter 3 describes our CUDA-CHiLL code transformation and generation compiler system. Optimization heuristics and their impact on performance, along with the search space of possible solutions and the pruning criteria is explained in Chapter 4. The core algorithms of the TSG are presented in Chapter 5. Chapter 6 presents performance results for three BLAS functions (matrix-matrix multiply, matrix-vector multiply and transposed-matrix-vector multiply), along with five other scientific and imaging kernels. Finally we discuss related work in Chapter 7 and conclude the thesis in Chapter 8.

Chapter 2

Background

This chapter provides background on the target architectures, benchmarks and compiler transformations that will be discussed in the remainder of the thesis.

2.1 GPU Architecture Features

The two architectures for this work are the older Nvidia GeForce GTX 280, and the newer Nvidia Tesla C2050 Fermi, described in Table 2.1. Both architectures are organized in a two-level parallelism hierarchy, with a number of streaming multiprocessors (SMs), each of which has a SIMD unit comprised of several cores. Synchronization between threads in an SM is supported by a barrier; synchronization between threads mapped to different SMs typically relies on atomic operations or a return to the host processor.

The GPU devices have a heterogeneous, and mostly software controlled, memory hierarchy. Both devices have a global memory that can be accessed by all cores on the

	GTX280	C2050
#SMs	30	14
cores/SM	8	32
Total cores	240	448
Peak (single precision)	933 GF/s	1.03 TF/s
Peak (double precision)	87 GF/s	515 GF/s
Global memory	1 GB	3 GB
Bandwidth	142 GB/s	144 GB/s
Shared memory/SM	16 KB	(up to) 48 KB
Registers/SM	16 K	32 K
"Texture" accesses	yes	yes
Constant memory	64 KB	64 KB
Data cache	0	(up to) 48 KB

Table 2.1: GPU target architectures .

device. Both devices also have a large register file consisting of 16K/32K registers, and each SM has a shared memory that threads in a block mapped to that SM can freely access. Read-only constant and texture data in global memory are cached for low-latency average access time. A data cache for global memory accesses is available on the C2050, but on the GTX280, the bulk of global memory accesses are typically not cached, with latencies on the order of hundreds of cycles.

A new architecture often requires a complete rewriting of the code, particularly if the new architecture is significantly different from previous targets for the software. While the fundamental execution model, memory structures and processor organization are similar for the two GPU devices discussed above, there are a few key differences highlighted in Table 2.1 that affect the code generation strategy.

The most significant difference is the presence of a data cache in the C2050. Further, the balance of the architectures is different. SMs in a C2050 have four times more cores than in the GTX280, which suggests that the number of threads per block should perhaps be larger on a C2050, in context of achieving higher performance. The configurable shared memory on the C2050 can be up to 3 times larger than the shared memory on the GTX280, which is consistent with being able to support more threads per block. The size of the register file per SM is also double in C2050 compared to the GTX-280 architecture. These architectural differences suggest that the optimization trade-offs may lead to different implementations for the same code, targeting the two platforms, which will be demonstrated in the performance results.

2.2 ***C***ompute ***U***nified ***D***evelopment ***A***rchitecture (CUDA)

A CUDA program describes a computation decomposition into a one or two-dimensional space of thread blocks called a *grid*, where a block is indivisibly mapped to one of the SMs. Each thread block defines a multi-dimensional space with up to three dimensions and maximum of 512 threads. A *kernel* code is executed for each point in the grid, providing a two-level parallelism hierarchy represented by this 5-dimensional space. Threads within a block run concurrently on the same SM in batches, called *warps*, under a SIMT[1]execution model.

With so many parallel threads simultaneously accessing memory hierarchy, effective utilization of the memory hierarchy has significant impact on performance. The programmer or compiler can *reduce memory latency* through *locality optimizations* that copy data into lower-latency portions of the memory hierarchy as compared to global memory on data that has *temporal or spatial reuse*. Whenever possible, registers provide the most ready low-latency access and the register file on the GPU is large (16K/32K registers per SM, on GTX-280 and C2050, respectively). By copying data into a thread-local variable and aggressive unrolling of loop nests, the high-level source code can expose reuse of local data that can be mapped to registers. Shared memory has latency comparable to registers (as long as there are no memory bank conflicts) for data that can be shared by all threads in a block. To *maximize memory bandwidth*, the memory controller will *coalesce* accesses to multiple data into a single memory transfer if the accessed data has spatial reuse within the size of a memory transfer [ULBmWH08]. Therefore, the programmer/compiler can order data accesses across neighboring threads (at least a half-warp in GTX-280 and a warp in Fermi architectures) so that global memory accesses are coalesced whenever possible. Another bandwidth optimization sequences shared memory accesses to avoid bank conflicts.

2.3 Benchmarks and Libraries

This thesis compares performance of automatically generated code by our system with implementations from manually tuned library and state-of-the-art compiler benchmark suites for BLAS, scientific, imaging and multimedia kernel domains. Our goal is to achieve performance, comparable to these manually tuned and automatically generated

[1]Single-instruction, multiple-thread (SIMT) is the name given to the Nvidia execution model where multiple threads in a SM execute concurrently in a lock-step manner.

benchmarks by the library and contemporary state-of-the-art compiler systems, respectively. This section provides an overview of these codes to be used throughout the thesis as examples, and in the performance evaluation of Chapter 6.

CUBLAS. CUBLAS is a basic linear algebra subprograms (BLAS) implementation from Nvidia running over Nvidia's CUDA runtime [Nvi08b]. It provides manually-tuned solutions for different linear algebra kernels, which provides us with a clean, self-contained computational resource to compare against. Over the course of this study, multiple versions of CUBLAS were released. The performance of the CUBLAS also stabilized and improved with time. We will present results from CUBLAS 2.2 and CUBLAS 3.2 to emphasize the need of systems like ours that can automatically generate optimized code and can consistently match or outperform the manually tuned code. We have compared against matrix-matrix multiplication, matrix-vector multiplication and transpose matrix-vector multiplication routines from CUBLAS, explained in detail in Chapter 6.

Benchmarks	Domain	Benchmark Source/ Performance Compared to
Matrix-Matrix Multiplication (GEMM)	Linear Algebra	CUBLAS
Matrix-Vector Multiplication (GEMV)	Linear Algebra	CUBLAS
Transpose Matrix-Vector Multiplication (TMV)	Linear Algebra	CUBLAS
NBody	Scientific Computation	PLUTO
Coulombic Potential (CP)	Scientific Computation	PLUTO
MPEG4	Multimedia Kernel	PLUTO
MRI-FH	Imaging Kernel	PLUTO
MRIQ	Imaging Kernel	PLUTO/PARBOIL
2D-Convolution (2D-Conv.)	Imaging Kernel	Compiler by [YXKZ10]

Table 2.2: Benchmarks set evaluated by CUDA-CHiLL based system.

PLUTO compiler benchmarks. We have also compared against a state-of-the-art contemporary compiler, PLUTO [BBK+08a, BRS10, LVM+10] by using the set of benchmarks that were provided to us by the group. These include, MRIQ, MRIQ-FH, NBody, MPEG4 and CP.

PARBOIL Benchmarks. The PARBOIL benchmark suite contains many applications which are better suited to performance measurement on a massively parallel platforms. The suite is provided by IMPACT Research Group at University of Illinois, Urbana Champaign as open source software [IMP07]. We have tested the MRIQ kernel from the suite and compared the performance with the solutions provided by the suite.

In addition to these benchmarks, we also compare against an imaging benchmark, 2D convolution, from [YXKZ10], which adds to the kernels from imaging domain we compared from PLUTO and Parboil benchmark suites.

2.4 Parallelizing Compiler Technology

This section presents well known compiler analyzes, used by traditional compilers for uniprocessors, shared-memory multiprocessors, vector architectures and memory hierarchy optimization. These analyzes lay the groundwork to generate GPU code by constraining computation decomposition possibilities and capturing memory access patterns.

2.4.1 Dependence Analysis

In any compiler system which modifies code structure and generates transformed code, dependence analysis needs to done to ensure the correctness of the transformed code. A fundamental approach to achieve correctness in the generated code is to create a dependence graph between different statements of a sequential program. In a dependence graph, any pair of statements is called a *dependence*. Dependences ensure that that the order of the memory read and write operations is preserved [AK02].

To analyze loop nest computations, we reason about the dependences within the iteration space of the loop nest. We use dependence distances to decide about the presence of dependences.

- If all distances are zero, then the dependence is within the same iteration of the loop nest (it is loop independent).

- If at least one distance is non-zero, it is a loop carried dependence.

We use dependence analysis within our GPU compiler for the following purposes:

- identifying loop transformation opportunities
- identifying opportunities for block-level and thread-level parallelism
- identifying constraints and opportunities for loop nest transformations.

In our compiler, the presence or absence of dependences determines safety of parallelism and other reordering transformations.

There are four basic types of dependences between two operations [AK02].

- Flow (True) dependence: A statement S2 is flow dependent on S1 (written) if and only if S1 modifies a resource that S2 reads and S1 precedes S2 in execution.
- Antidependence: A statement S2 is antidependent on S1 (written) if and only if S2 modifies a resource that S1 reads and S1 precedes S2 in execution.
- Output dependence: A statement S2 is output dependent on S1 (written) if and only if S1 and S2 modify the same resource and S1 precedes S2 in execution.
- Input dependence: A statement S2 is input dependent on S1 (written) if and only if S1 and S2 read the same resource and S1 precedes S2 in execution. It is not required for safety, but is helpful for locality.

2.4.2 Locality Analysis and Optimizations

Locality refers to accessing the same or nearby data close together in time, in fast memory. Many scientific applications access data in a certain pattern, which may allow for certain optimizations to improve performance.

Locality within a loop nest. A data reference is said to have *locality* if the same data or nearby stored data are accessed from the memory hierarchy level of interest by multiple iterations of a loop nest.

Types of Locality.

- Spatial Locality: the increased probability of accessing a nearby memory location after a certain memory access. Moving a chunk of data, including the required value along with some of the neighbors to a faster memory in the hierarchy, may help in achieving high performance.

- Temporal Locality: a special case of spatial locality, where the same memory location might be accessed again in the near future, following an access. This property allows the executing program to keep the accessed data in a faster memory in the hierarchy for a longer stretch of time, to achieve high performance.

Locality Optimization. In our system, we analyze the array references to identify reuse patterns, among threads and across threads. High performance can be achieved by copying data reused across threads into shared memory and reused inside threads, into registers.

The tiling transformation has been used in our system as a technique for achieving parallelism and locality optimization. Other contemporary systems [BBK$^+$08a] have also demonstrated the use of tiling for both purposes. For data locality, tiling helps divide the iteration space into tiles which can be copied into smaller but faster memory.

An important feature of CUDA execution model is the ability to batch a number of memory accesses from global memory together in a single memory read, called *global memory coalescing*. This feature in GPUs can be exploited to bring in spatially local data from the global memory in as few as a single memory operation, for a set of threads called *warps* (16 (half-warp) or 32 threads, depending on the compute capability of the CUDA device) [Nvi08a]. Memory coalescing is probably the single most important memory access optimization, that needs to be exploited to achieve high performance and is discussed in Chapter 4.

2.4.3 Loop Optimizations and Transformations:

Loop optimization enables compilers to identify parallel computations, enabling code optimizations and improved cache performance. Loop nests take much of the execution time in many computations. Optimizing loop nests to take advantage of parallelization opportunities and exploit locality results in faster execution.

2.4.4 Loop Transformations

A loop transformation reorders the execution of statements in a loop. A loop transformation is said to be safe if it preserves the dependences of the original program in the modified program. In theory, each transformation is applied to achieve performance improvement, but some transformations may only be performed as a prerequisite to another transformation, which yields benefits. Thus sometimes the individual transformations are beneficial and at other times a combination of multiple transformations is needed.

Common loop transformations used in our system.

- Loop permutation: This optimization reorders the loops in a nest, changing the order in which memory is accessed. Such a transformation can improve locality, depending on the array's layout. Different loop orders may also enable other loop transformations or a desired loop transformation may need loop permutation as a prerequisite operation.

- Loop tiling/blocking: Loop tiling reorganizes a loop to iterate over blocks of data sized to fit in the cache. In our system, loop tiling is used to achieve parallelism and locality optimization. A loop nest's iteration space can be tiled into blocks and threads to set up the launch configuration for a kernel. Also, tiling can be used to reduce the data footprint so that it fits into smaller and faster memory in the hierarchy.

- Loop unrolling: Unrolling replicates the body of the loop multiple times, in order to decrease the number of times the loop condition is tested and the number of branches, which may degrade performance by impairing the instruction pipeline. Completely unrolling a loop eliminates all overhead (except multiple instruction fetches and increased program load time), but requires that the number of iterations be known at compile time. We use explicit unrolling of loops in order to have constant indices to the array references which can be mapped to registers. For some kernels, we differentiate ourselves from other contemporary solutions by the use of aggressive unrolling.

Data Placement Optimizations. Data related optimizations try to improve chances for locality, by moving data to faster memories or by increasing the memory bandwidth by using parallel hardware to access data in global memory. We target all the memories

- Data Copy (shared memory): We use data copy to move data from a slower memory to a faster one [Che07]. We use it for copying data to shared memory from the slower global memory in the context of a GPU. Data copying to a faster memory helps in improving performance by lowering access times and also by simplifying addressing. Also, it lowers the number of conflict misses in a cached architecture, by copying a tile of data into a faster memory. A copy to shared memory from global memory is available to all threads in a particular block.

- Data Copy Private (register): A specialized data copy is the copy of data to local memory of a thread, which is the register file. We use the fastest memory in the hierarchy to further improve performance.

In addition to local and shared memory, every thread has access to two more read-only spaces. These are texture memory and constant memory. Fermi architectures are also equipped with a data cache of size varying from 16k-48k. Our system targets texture and constant memory spaces too, with similar data copy transformations as to shared memory and registers.

- Data Copy (texture memory): A data copy to texture memory is achieved through a bunch of directives in the generated code, for read-only arrays. It allows for a specific array to be read through a parallel hardware used for texture reads, thus increasing the memory bandwidth.

- Data Copy (constant memory): A data copy to constant memory is a similar process as to texture memory. The size of the array is limited by the maximum constant memory size, as reported in Table 2.1.

2.5 The Polyhedral Framework

The polyhedral model is a geometrical representation for loop nest computations. It uses linear algebra and linear programming techniques to analyze programs and apply code transformations. The polyhedral model [Bas04] handles a wider class of programs and transformations than the uni-modular framework, which uses a single matrix to describe the result of many transformations combined [Ban93, Ban90, WL91]. The executions of a set of statements within a possibly imperfectly nested loops is seen as the union of a set of polytopes representing the executions of the statements. Affine transformations are applied to these polytopes, producing a description of a new execution order. The boundaries of the polytopes, the data dependences, and the transformations are often described using systems of constraints. This approach is often referred to as a constraint-based approach to loop optimization. For example, a single statement within an outer loop 'for i := 0 to n' and an inner loop 'for j := 0 to i+2' is executed once for each (i, j) pair such that $0 \leq i \leq n$ and $0 \leq j \leq i+2$.

2.6 Summary

This chapter outlines the architectural features of the two target GPUs and the characteristics of the programming model used to program them. We focus on two particular

Nvidia GPUs in this study, which belong to two architecturally different generations of Nvidia GPUs. CUDA is the programming framework used to program them. An overview of the benchmarks we have evaluated with our system was presented. We have discussed background of important compiler functions like dependence analysis, locality analysis, and loop transformations and optimization prior to a more detailed discussion in coming chapters of their utilization in our system. Our system uses a code transformation and optimization framework which is based on the polyhedral framework and is explained in detail in the next chapter. This mathematical way of handling code transformation helps in achieving correct code generation for a range of different code variants.

Chapter 3

System Overview

In this chapter we present the organization of our compiler system. Figure 3.1 shows the major sub-systems, describing a chain of processes which directs both parallel computation partitioning and data staging, generates transformation strategies, and use autotuning to evaluate these and find a high performance GPU (CUDA) code solution.

3.1 Main Components of the Compiler System

This section presents the overview of the compiler system, shown in Figure 3.1 which includes two components that are focus of this thesis.

- Transformation strategy generator (TSG)
- Autotuning

This core part of the system is the focus of this thesis. However, it does not work as a standalone system. It utilizes important sub-systems which are,

- Composable High-Level Loop Transformation Framework (CHiLL) [Che07]
- CUDA-CHiLL [Rud10]

The input to the compiler system is sequential C code and the output are the best performing transformation strategy along with the best performing GPU (CUDA) code.

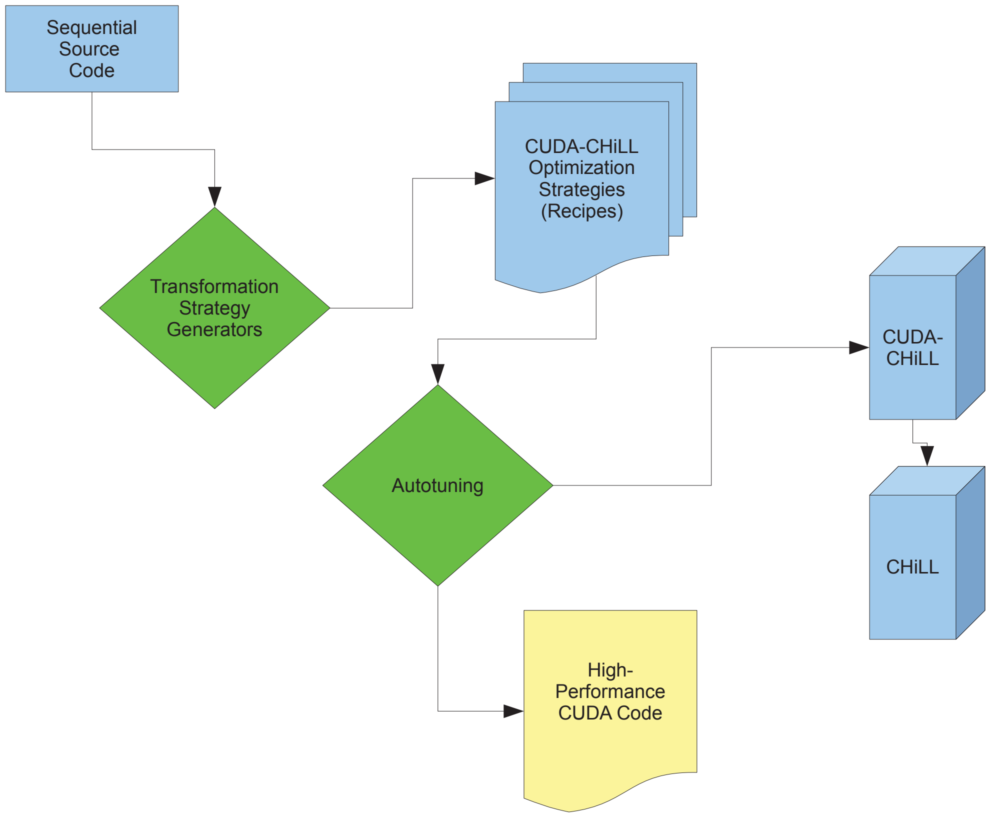
Sequential Source Code
Transformation Strategy Generators
CUDA-CHiLL Optimization Strategies (Recipes)
Autotuning
CUDA-CHiLL
CHiLL
High-Performance CUDA Code

Figure 3.1: Overview of system for high performance GPU (CUDA) code generation.

3.1.1 Transformation Strategy Generator (TSG)

The major contribution of this thesis is the Transformation Strategy Generator (TSG) that analyzes the input code, derives a set of different parameterized optimization strategies, and thus logically expresses a search space of possible high performance implementations of a computation. The TSG uses analysis to decide on the code transformations while several features both related to architecture and the computations help cap the size of the possible search space of solutions. The output of the algorithms in TSG is not code, but rather a *transformation recipe* that expresses an optimization strategy as a sequence of composable transformations [DBR+05, HCS+09].

3.1.2 Autotuning

The other important contribution of this thesis is the mechanism of autotuning, which expresses the search space through individual transformation recipes, navigates the search space efficiently and finally finds the best performing solution for a given computation on a given architecture. The strength of this autotuning mechanism is the flexibility of independently autotune for the compiler optimizations and their respective parameters for values, to transform a given computation.

3.1.3 ***C***omposable ***Hi***gh-***L***evel ***L***oopt Transformation Framework (CHiLL)

CHiLL is a loop transformation framework that efficiently transforms and generates code for complex loop nest applications. We will discuss in the next section the specific CHiLL commands and their use, shown in Table 3.1. CHiLL operates most effectively on sequential code restricted to an affine domain, where loop bounds and subscript expressions are linear functions of the loop index variables. The CHiLL transformation algorithms will generate correct code when dependence conditions are satisfied. Such dependence information is kept updated during each step of the transformation process.

The script interface allows the users to compose and try many different combinations of the transformations, thus providing a flexible platform for tuning transformation solutions [CCH08]. These features make CHiLL a key component of our compiler system, providing the technology to support autotuning. We will discuss in the next section, the specific CHiLL commands and their use, shown in Table 3.1, in our system.

3.1.4 CUDA-CHiLL

CUDA-CHiLL is a source to source compiler transformation and code generation framework for the parallelization and optimization of computations expressed in sequential loop nests for running on many-core GPUs. This system uniquely uses a complete scripting language to describe composable compiler transformations that can be written, shared and reused by non-expert application and library developers. CUDA-CHiLL is built on CHiLL, which is capable of robust composition of transformations while preserving the correctness of the program at each step [Che07, TCC$^+$09, Rud10].

The rest of the chapter first explains the concept of transformation recipes, CHiLL commands constituing the low level recipes and CUDA-CHiLL followed by an overview of the core components TSG and autotuning, explained in detail in Chapter 5.

3.2 Transformation Recipe/Strategy

The composition of the CUDA-CHiLL commands is called a *transformation recipe* or *transformation strategy*.It lists high level transformations in CUDA-CHiLL which may be accompanied by low level CHiLL commands. Thus a recipe is an individual strategy to modify the code structure. The transformation recipe also is the interface for application or library developers, to interact with the system directly. We use the terms *transformation recipe* and *transformation strategy* interchangeably in the rest of this thesis.

3.2.1 CHiLL Transformations

Each line in a transformation recipe is a single transformation to be applied to the input code. We call each of these transformation commands. Table 3.1 describes the relation of several low level CHiLL commands to the high level CUDA-CHiLL transformation commands. The table also shows how different CHiLL commands are composed to form higher-level CUDA-CHiLL commands. However, in the implementation of these interfaces, many of the constituent transformations were used multiple times, to achieve the effect of one correct high level transformation operation. For example, the CHiLL

CHiLL Command	Used by CUDA-CHiLL Command	Description
tile	tile_by_index, copy_to_shared copy_to_registers	Tile a loop with a given tile size. Specify an index variable for the new control loop and a second index variable to optionally rename the original index variable of the tiled loop.
permute	permute	For a specified statement, reorder loops in a loop nest (the iteration space of the statement.) The permutation order is specified by a list of loop index variables.
datacopy	copy_to_shared	For a specified statement, starting loop level and array variable, copy array data that is accessed within the starting loop level to a smaller dimensional structure. Optionally annotate the new data structure with `__shared__` to specify a copy to shared memory.
datacopy _privatized	copy_to _registers	Similar to `datacopy` but used to copy data private to a thread and thus does not have an option to flag for shared memory.
unroll	unroll_to _depth	Exposes ILP and register reuse to the backend nvcc compiler, reduces loop overhead and can provide integer speedups on GPUs.

Table 3.1: Description of common CHiLL commands matching higher level CUDA-CHiLL commands.

command *tile* is used in the composition of most of the higher level CUDA-CHiLL commands.

3.3 CUDA-CHiLL: Parallelization and Data Staging

CUDA-CHiLL, just like CHiLL, provides a powerful scripting interface which allows the users to define transformations at a much more abstract level than CHiLL. Usually, the CUDA-CHiLL abstractions incorporate many CHiLL commands in a single high-level command. Thus, it provides a much more accessible way of programming the GPUs by hiding the low level constructs and details of the GPU programming models. An important feature that CUDA-CHiLL inherits from CHiLL is the capability of the framework to transform sequential code for any given architecture with appropriate modifications to the code generation part of the system. This section will list in detail the CUDA-CHiLL constructs, their individual composition from CHiLL commands and the parameters driving their operations.

3.3.1 High Level CUDA-CHiLL Commands

Our compiler system expresses an optimization strategy as a sequence of composable transformations, which has been followed as a way of laying out the optimization strategy in multiple research works [DBR+05, CWX+11, HCS+09]. Our system expresses the

Command	Example Parameter List	Description
tile_by_index	{"i","j"}	The index variables of the loops that will be tiled.
	{TI,TJ}	Variables representing the respective tile sizes for each index variable.
	{l1_control="ii", l2_control="jj"}	Specifies control loop variable names and optionally renames tile loop index variables.
	{"ii", "jj", "i", "j", "k", "l"}	Final order of nested loops with updated loop index names.
cudaize	"conv_GPU"	The name of the kernel function
	{a=(N+M)*(N+M), b=M*M, c=(N+M)*(N+M)}	The data sizes of the arrays if not statically determinable.
	{block={"ii"}, thread={"jj"}}	Specifies block and thread indices. The bounds for these loops are used to derive grid dimensions.
copy_to_registers	"kk"	The loop level, given as an index variable, to place the copy.
	"b"	The array variable to be copied.
copy_to_shared	"tx"	The loop level, given as an index variable, to place the copy.
	"b"	The array variable to be copied.
	-16	Ensure the last dimension of the temporary array are coprime with 16 to pad.
copy_to_texture	"b"	The array variable to be accessed from texture memory.
copy_to_constant	"b"	The array variable to be accessed from constant memory.
unroll_to_level	1	Unrolls all statements up to specific level from innermost loops outwards. Stops unrolling if it encounters a CUDA thread index.

Table 3.2: Description of important high level commands in CUDA-CHiLL.

code transformations at a very high level using the CUDA-CHiLL constructs described in Table 3.2. When necessary, slightly lower level CHiLL transformation such as tiling individual loops, unrolling a specific loop and so on, can also be performed. CUDA-CHiLL takes these high level transformation specifications tailored to CUDA code generation and generates the specific set of compiler transformations that must be composed to produce the CUDA code. This abstraction layer is written in the embeddable scripting language Lua [IdFF96], which permits the compiler developer to concisely express the transformation algorithms.

3.4 Transformations for GPU Code Generation

In this section, we briefly explain the process of generating optimized CUDA code from sequential code through the high level CUDA-CHiLL commands, as shown in Table 3.2.

3.4.1 Computation Mapping using Tiling

We observe that an Nvidia GPU is a *tiled* architecture, with each streaming multiprocessor (SM) representing a separate tile with its own private memory, similar to RAW [TKM+02]. Therefore, to generate code that executes in parallel across SMs, we need a computation decomposition where each SM operates on localized data. Then, each thread also operates on mostly independent data.

Subdividing the iteration space of a loop into blocks or *tiles* with a fixed maximum size has been widely used when constructing parallel computations [Wol89, KM92]. The shape and size of the tile can be chosen to take advantage of the target parallel hardware and memory architecture, maximizing reuse while maintaining a data footprint that meets memory capacity constraints. Sometimes referred to as *loop blocking*, *tiling* involves deconstruction of an iteration space into a control loop and tile loop. Given an iteration space of size N and a tile size of TX the tile loop will iterate over a maximum space defined by the tile size (TX), while the control loop then has N/TX steps (when TX divides N evenly).

Figure 3.2 represents a simple computation partitioning and CUDA code generation for the 2D convolution code shown in Figure 3.2(a). The two tile commands in the recipe of Figure 3.2 (b) result in the tiling of both the i and j loops of the sequential computation, as shown by the automatically generated code of Figure 3.2 (c) resulting from applying the recipe to the sequential code and printing the intermediate result at line 5.

After tiling, the *cudaize* command specifies the name of the generated kernel function, the mapping of one or two loop levels to block indices in the grid dimension, and the mapping of up to three loops to thread indices. The effect of the *cudaize* call on the generated code is to allocate storage and marshal inputs for the CUDA kernel, and select the loops from the transformed loop nest that will be mapped to grid and thread dimensions. These tile controlling loops are effectively removed from the code. The resulting automatically generated CUDA code is shown in Figure 3.2 (d).

```
void seqConv(float c[N+M][N+M], float a[N+M][N+M],float b[M][M])
{
  int i,j,k,l;
  for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
      for (k = 0; k < M; ++k)
        for (l = 0; l < M; ++l)
          c[i][j] = c[i][j] + a[k+i][l+j] * b[k][l];
}
```

(a) The 2D-convolution sequential source code.

```
TI=16
TJ=16
tile_by_index({"i","j"}, {TI,TJ},
   {l1_control="ii", l2_control="jj"},
   {"ii", "jj", "i", "j" , "k" , "l"})
cudaize(‘‘conv_GPU’’, a=(N+M)*(N+M), b=M*M, c=(N+M)*(N+M),
   block=‘‘ii’’,‘‘jj’’, thread=‘‘i’’,‘‘j’’)
```

(b) A very simple transformation recipe.

```
for (ii = 0; ii < 64; ii++)
  for (jj = 0; jj < 64; jj++)
    for (i = 16 * ii; i < 16 * ii + 16; i++)
      for (j = 16 * jj; j < 16 * jj + 16; j++)
        for (k = 0; k < M; ++k)
          for (l = 0; l < M; ++l)
            c[i][j] = c[i][j] + a[k+i][l+j] * b[k][l];
```

(c) The results of tiling the sequential code.

```
__global__ void conv_GPU(float (*c)[1024], float (*a)[1024],
    float (*b)[16])
{
  int bx = blockIdx.x; int tx = bx*blockDim.x+threadIdx.x;
  int by = blockIdx.y; int ty = by*blockDim.y+threadIdx.y;
  for (k = 0; k < M; ++k)
    for (l = 0; l < M; ++l)
      c[tx][ty] = c[tx][ty] + a[k+tx][l+ty] * b[k][l];
}
```

(d) The resulting CUDA kernel, invoked as gpuConv$\langle\langle\langle 64, 64\rangle\rangle\rangle$(...)

Figure 3.2: A simple tiled and CUDAized 2D Convolution.

3.4.2 Data Staging in Memory Hierarchy using Data Copy

Multiple transformations are used for the complex data copy operations, to enable movement of data to faster memory levels of the hierarchy. We present them in the following subsections, to individually study their role and effects on data staging operations.

Data Copy. The main operation for data staging is the copying of data into different levels of memory hierarchy. CUDA-CHiLL specifies separate data copy commands, specifying copy of data in register, shared memory, texture memory and constant memory. All of these copy commands work on the name of the data structure given as parameter, but only two, the copy to registers and copy to shared memory are supplied with the intended loop level positions for the copy loops. For texture memory and constant memory copies, only the names of the data structures is enough to use the respective memories. To support CUDA code generation, we augment pre-existing data copy with a new capability that supports parallel code generation and GPU-specific memory structures.

When copying into shared memory, the copy must be properly synchronized to avoid race conditions with other threads. Further, a private copy into registers copies a data footprint that is private to individual threads. The code generator must also use the right type annotation, `__shared__` for shared memory and a local variable declaration for registers. Because of their robust underlying implementation in a polyhedral framework, these *datacopy* transformations ensure correctness by understanding the data dependences between loop statements, deriving conservative data footprints automatically for a specific loop nest level, and handling cases of unevenly divisible loop ranges. This again simplifies an error prone optimization step in achieving high performance CUDA kernels.

Tiling for Data Copy. In addition to its role in setting up parallelization, tiling also reorganizes computation within or across threads in the program so that the data footprint fits within a specific level of the memory hierarchy. Data destined for shared memory must be explicitly copied to/from the device global memory. Similarly, data locally declared within a thread program will usually be placed in registers by the `nvcc` backend compiler, but such data may require explicit copies to/from global or shared memory. Therefore, we couple tiling with explicit copying of data and associated synchronization to perform the *data staging* into shared memory or registers.

When managing registers, we must also explicitly unroll the tiled loops so that accesses to local array data will have constant indices that can be mapped to specific registers. CHiLL supports loop unrolling through a combination of iteration space transformations and statement rewriting.

3.5 Integration Interface for TSG and CUDA-CHiLL

We have integrated the sub-systems described in this chapter to form an end-to-end system. There are two separate interfaces we have developed as part of this work. The core part of the CUDA-CHiLL interface to CHiLL, is written in C++, and provides the CHiLL transformations, the data structures representing the current state and GPU related transformation and code generation capabilities.

The main contribution of our work to the Lua interface is limited to the handling of data copy operations for registers and shared memory and unroll to a certain depth level for the whole code structure. The goal, in the case of data staging operations for registers and shared memory, is to create data copy loops at the correct locations in the generated code.

(1) Computation Partitioning: The computation partitioning interface (added in this list for completion purposes) remains the same as was provided by CUDA-CHiLL, with its basic functionality of applying tiling transformation to modify loop nests to create loops carrying parallelism.

(2) Data Copy Interfaces: The data copy interfaces were the major contribution of the integration code. The interfaces initially provided for the shared memory copy and register copy were basic and limited to a few BLAS computations only. The shared memory data copy interface had to be re-written to incorporate the scenarios which arose from new computations. New code structures were written

for using the existing functionality along with the additional code handling the new computations. Similar additions were needed in the register copy, however a major part of the code was carried over.

(3) Unroll Interface: For unroll, it controls the number of individual CHiLL unroll commands issued for specific loops, by unrolling not only the original loop but also any additional clean up loops created by the transformations.

These functionalities have either been modified or completely re-written to allow for more flexible handling of different transformation strategies in different types of problems.

As a whole, the interface interprets the current loop structure and identifies, labels and modifies it, driven by the CUDA-CHiLL command parameters. Thus, for example, one parameter in the *cudaize* command of Table 3.1, identifies and labels the loops designated for thread parallelization, to be later interpreted by the code generator. An *unroll_to_depth* CUDA-CHiLL command searches and unroll loops in the entire structure, thus generating multiple unroll CHiLL commands for different loops.

3.6 Summary

In this chapter, we presented an overview of the major components of the system. Much emphasis was given to the explanation of the underlying sophisticated compilers systems called CHiLL and CUDA-CHiLL. The main contribution of this thesis are the TSG and autotuning parts of the system, which are going to be discussed in detail in the coming chapters.

Chapter 4

Optimization Heuristics, Search Space and Constraints

This chapter presents the optimization heuristics, defines the search space of unique optimization combinations and the use of heuristics to constrain the size of the search space. The process of defining a search space is explained through the matrix-matrix multiplication example.

4.1 Optimization Strategy

To develop our optimization strategy, we considered manual tuning strategies for GPUs [VD08a, BG09, BCI$^+$08], compiler approaches for combining parallelism with locality optimization (*e.g.,* [KM92]) and memory hierarchy optimization strategies that compose multiple optimizations [CCH05, MCT96, NTD10]. The key insight from the algorithm in [CCH05] is to use compiler transformations to control placement of data in different levels of the memory hierarchy according to the amount of reuse. The data with maximum reuse is placed in the fastest portion of the memory hierarchy, and tiling is used to match the data footprint of the reused data to the capacity of the different storage levels. We consider three main levels of memory hierarchy in the target GPU: register, shared memory and global memory. In addition to them, we consider constant and texture cache which helps in exploiting additional memory space and allowing for additional bandwidth due to separate access hardware (in the case of texture memory). Unlike a conventional cache-based memory hierarchy where different memory hierarchies have an order of magnitude difference in size, the GPU's register file capacity is comparable to that of shared memory, and latencies are in the best case equivalent. Since we are generating parallel code, we use shared memory to exploit reuse across threads and registers to exploit reuse within a thread.

4.2 Optimization Heuristics

This section describes the key optimization heuristics, used by our system in striving for performance. "These heuristics constrain the optimization strategies considered and limit the possible values for the optimization parameters.

4.2.1 Computation Partitioning Heurisitics

Dependences and Parallelization. We permute the loops in the nest so that any loop carrying a dependence is within a thread or a thread block. In the latter case, thread synchronization must be inserted. Loops representing blocks should not carry dependences, as this would require costly global synchronization. Different levels of subloops within a loop nest can be parallelized as long as they use the same thread block size and proper synchronizations are inserted.

Global Memory Coalescing. All data is initially copied into global memory. Therefore, we select a loop order such that the x dimension for the thread index is linearly accessing the bulk of its data in global memory, resulting in coalesced access. If global memory coalescing is not possible due to interference with another array accessed in a different order, an array that is reused across threads may be copied by different threads into shared memory in a coalesced order, and accessed directly from shared memory.

4.2.2 Data Placement Heuristics

Shared Memory copy and Bank Conflicts. Data shared across threads, either as a result of the global memory access coalescing optimization above, or through significant inherent reuse, is placed in shared memory. Shared memory accesses need to avoid bank conflicts along the thread index, and 2-dimensional arrays that are loaded into shared memory linearly along one dimension and accessed linearly in another dimension will require padding of one of the dimensions to avoid shared memory bank conflicts.

Reuse in Registers. Registers provide low latency storage local to a thread, and data that is reused within a thread should be copied into registers. Due to the large register file size, tile sizes for register tiling are also good candidates for partitioning the computation across streaming multiprocessors, thus avoiding an additional level of tiling and overhead.

Map Read-Only Data to Texture Memory. Data that is read-only within a thread may benefit from being accessed through a special texture memory designation, which uses dedicated hardware to fetch the data, and in the GTX-280, permits data caching of global memory accesses. The data for which texture accesses are beneficial must be copied into either registers or shared memory and reused or else the texture access is not profitable.

Map Read-Only Data to Constant Memory. Data that is read-only within a thread and possible reuse across blocks may benefit from being accessed through a special constant memory copy, which uses dedicated hardware to fetch the data, and in the GTX-280, permits data caching of global memory accesses. It also may allow tiling of execution of the kernel at the host. However, the maximum benefit from the constant memory copy may only be gained if the input data is less than the maximum size of constant memory.

4.2.3 Code Transformation Heuristics

Unroll Loops Aggressively. Loop unrolling exposes instruction-level parallelism and register reuse to the backend nvcc compiler, reduces loop overhead, and can provide integer speedups on the GPU. Because of the large register file and the small number of threads typically selected by our algorithm, we can fully unroll entire 2-dimensional tiles without exceeding register capacity.

Our compiler structure facilitates exploring a rich search space which subsumes that of previous GPU compilers [BBK+08a, BRS10, LVM+10, CWX+11]. To create a manageable search space, we employ the above discussed compiler heuristics to focus on candidate implementations that are likely to yield high performance.

The coming section articulates the optimization search space along with constraints governing it. These constraints stem from the heuristics discussed earlier in the chapter. This discussion about the search space also serves as a road-map for the next chapter where the algorithms are presented in detail. We explain the search space in the context of our example from linear algebra kernels, matrix-matrix multiplication. Because the compiler can explore multiple optimization strategies, insights from the experiences of manually tuning this code were integrated into our compiler implementation [VD08a, NTD10].

4.3 Defining the Search Space

The section presents the search space of possible optimizations strategies, that is essentially created by combining the optimizations for computation decomposition and data placement, with multiple values of the optimization parameters. Our system tries to avoid the local maxima by looking at a larger set of possible combinations of optimization decisions and parameters. The mechanism is presented in separate algorithms in Chapter 5.

4.3.1 Optimization Decisions

The first set of optimization decisions is related to management of the computation workload. It is crucial to strike a balance between the entities which are assigned the sub-computations such as blocks and threads per block; and the available resources such as number of registers, amount of shared memory, maximum number of blocks or threads per block.

1. **Computation Partition:** *Decisions on the structure and number of blocks and threads per block, realized using permute and tiling.* The key considerations to take computation partitioning decisions are described below:

   (a) Number of dimensions of blocks and threads are usually made to match the dimensions of key data structures.

   (b) The candidate loops chosen to select for threads and blocks are based on key information regarding loop dependences, global memory coalescing and reuse.

   (c) The candidate thread dimensions are created from a single loop or from multiple loops (determines data partitioning across threads).

   (d) The data distributions per thread can be cyclic vs. block distributions in all dimensions (affects global memory coalescing and memory system).

   The result of this part is a possible set of tiling and permutation commands, which represent possibly multiple computation decomposition. A special command called *cudaize* would identify the loops which are designated as blocks and threads. These loops wont appear in the final generated code.

   In case of multiple computation decomposition strategies, they are represented by a unique combination of tiling, permutation and cudaize commands. These strategies are combined with optimizations from the data placement decisions, some of which may not be valid, resulting in a complete optimization strategy.

2. **Data Placement:** *Decisions on the placement in memory hierarchy, of data from data structures in the computation; realized through copy commands for specific memory level.* The key considerations to take data placement decisions are described below:

(a) Read-write data structure with reuse within a thread should try to use registers up to capacity limits (may require additional tiling to match register footprint or use existing tile).

(b) Read-write data structure with reuse across threads should try to use shared memory (may require additional tiling to match footprint to parallel threads).

(c) Read-only data structure with reuse across threads may try constant memory (may require additional tiling at the host to fit data into constant memory).

(d) Read-only data already mapped to shared memory may try to use read-only texture memory.

The result of this part of optimization decision making may lead to multiple copy commands, directed at copying data to selected levels of memory hierarchy. The system specifies and number of copy commands explained in Table 3.2. The number of possible decisions have a tendency to grow exponentially with linear increase in the number of data structures. Multiple data structures can be mapped to a single level of memory hierarchy, and a single data structure can be decided to go into different levels of memory hierarchy. In the first case, the search space will get populated with the strategies, placing each of the data structures and the unique combinations among themselves, in the decided memory level. In the second case, a data structure may be copied to shared memory, but by making use of read-only texture memory, to make use of dedicated hardware allowing for extra bandwidth. Here also the number of unique combinations of optimizations include the individual copies as well as their unique combinations with each other.

This set of optimization decisions are combined with those made in the previous part of computation decomposition to lay out each possible combination of optimizations in a separate strategy and describe the search space of optimization decisions.

4.3.2 Optimization Parameters

The second source contributing to the search space of unique optimization combinations is the set of optimization parameters that are specific to optimizations selected for computation decomposition and data placement. These parameters can have multiple values, driven by various constraints.

Parameter Values. *Selection of parameter values that govern number and identity of threads and blocks, data footprint in different levels of memory hierarchy, and unrolling to a certain depth in loop nests.* The optimization parameters define some of the key ingredients of a high performance GPU solution. Some are given below:

1. Number of blocks computing a sub-computation.
2. Number of threads for each dimension in a block.
3. Granularity of computation within a thread.
4. Amount of data to be placed in different memory hierarchy levels such as shared memory or registers at a time (may require additional tiling).

The optimization parameters, whether they are related to computation partitioning or to data placement, are treated the same. The initial values and the step between two values, given to these variables start with a value equal to a warp or a half warp (i.e. 32 or 16). The maximum value is limited by the allowed number of threads per block. However the number of these values are pruned by the tile size selection algorithm, explained in Chapter 5.

These unique combinations of optimizations decisions can be combined with different values of the optimization parameters, possibly creating a huge search space of strategies. However, a key autotuning methodology, explained in the algorithms in Chapter 5, restricts these kind of combinations, and looks at the two logical parts of search space separately, allowing a manageable number of optimization strategies to form part of the search space.

4.4 Case Study: Matrix-Matrix Multiplication

To illustrate important differences in the search space explored by our compiler, we use the examples in Figure 4.1, the code generated by our tool for matrix multiply for the two architectures, starting from the sequential code in Figure 4.1(a).

We first consider the automatically-generated transformation recipe and the generated code in Figures 4.1(b) and (d) for the GTX-280 and for Tesla C2050, respectively. In both examples (Figures 4.1(b) and (d)), loops i and j are selected for block loops, as they carry no dependences, and having two block dimensions matches the 2-dimensional array accesses in the computation. Matching the dimensionality of the largest-dimensioned data structure to the dimensionality of threads and blocks is a heuristic in our system which prunes many alternate strategies, and is consistent with other compilers and manually-written code.

In the GTX-280 versions for 1a and b, we tile the tile loop for i further to create two dimensions of threads, while maintaining a per-thread, partial column (block) data partitioning for output c, and a column and row partition for inputs a and b, respectively. For the Tesla C2050, to support 1a and 1b, at lines 3 and 4 of the script we tile both i and j loops to create threads (and also impact data footprint as described below). This results in a per-thread data layout for c that is two dimensional. To achieve global memory coalescing on accesses to this two-dimensional object, we permute the tile controlling and tile loops for j (see the reordering in line 4). This gives us a two-dimensional cyclic partition for c. The 2D cyclic partition of c subtly impacts the order in which c is accessed from the global memory, which in turn may affect bank conflicts in global memory [YXKZ10] and smooths out the performance for different problem sizes [NTD10]. In both versions at line 2, the script tiles the k loop to be placed outside the potential thread loops, both for exploiting reuse of c in registers at thread level and b in shared memory inside the block. The subsequent copy at lines 7 and 10 of Figure 4.1(b) needs to identify the footprint of c to be placed in registers, and this corresponds to the data accessed within the thread program which will arise from loop i. Array b is targeted for shared memory in both versions because it carries reuse and requires shared memory to support coalesced access. The *cudaize* command in both scripts triggers CUDA code generation; it appears at this point in the script because prior statements set up the computation partitioning and loop structure, and subsequent statements refer to transformations that occur inside the kernel program (unrolling).

The result of Figure 4.1(b) closely matches the SGEMM code in [VD08a] except for a different parallel mapping scheme for the shared memory copy code, and the use of texture memory loads for b. The Tesla C2050 version varies in a few ways. In addition to the differences in data distribution, the chosen recipe puts both inputs into shared memory, given the larger shared memory size available on the Tesla, favoring the larger SM over L1 cache. Also, one or two of the inputs (depending on problem size) is assigned to the texture memory to further accelerate loading data into shared memory. The resulting code in Figure 4.1(e) is similar to that of the MAGMA library [NTD10], but we do not perform a *prefetch* of input data into dummy variables to hide the latency of memory accesses, and for larger matrix sizes (> 1k), we only access b through texture memory.

The single precision (SGEMM) code, not shown in Figure 4.1, uses much larger tiles on the Fermi than the GTX-280 (TX=TY=96 on Fermi and TX=64,TY=16 on GTX-280); by doing so, we can control the data footprint as well as the computational decomposition suitable for the architecture. It is interesting that such similar scripts generate such differences in code.

4.5 Summary

This chapter presented the optimization search space we explore and the heuristics we employ to constrain this space. It is an important goal to restrict the size of the search space of unique implementations to keep its navigation practical.

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      c[j][i] = c[j][i] + a[k][i] * b[j][k];
```

(a) Matrix multiply source code (BLAS)

```
tile_by_index({"i","j"},{TI,TJ},{l1_control="ii",l2_control="jj"},
                    {"ii","jj","i","j"})
tile_by_index({"k"},{TK},{l1_control="kk"},
                    {"ii","jj","kk","i","j","k"},strided)
tile_by_index({"i"},{TJ},{l1_control="tt",l1_tile="t"},
                    {"ii","jj","kk","t","tt","j","k"})
cudaize("mm_GPU",{a=N*N,b=N*N,c=N*N},
              {block={"ii","jj"},thread={"t","tt"}})
copy_to_shared("tx","b",-16)
copy_to_texture("b")
copy_to_registers("kk","c")
unroll_to_depth(2)
```

(b) CUDA-CHiLL script for SGEMM on GTX-280.

```
float P1[16];
__shared__ float P2[16][17];
bx = blockIdx.x, by = blockIdx.y;
tx = threadIdx.x, ty = threadIdx.y;
P1[0:15] = c[16*by:16*by+15][tx+64*bx+16*ty];
for (t6 = 0; t10 <= 1008; t6+=16) {
  P2[tx][4*ty:4*ty+3] = tex1Dfetch(texbRef,1024*
    ((16*by+4*ty:16*by+4*ty+3) + (tx+t6)));
  __syncthreads();
  P1[0:15] += a[t6][64*bx+16*ty+tx]*P2[0][0:15]
  P1[0:15] += a[t6+1][64*bx+16*ty+tx]*P2[1][0:15]
  ...
  P1[0:15] += a[t6+15][64*bx+16*ty+tx]*P2[15][0:15]
  __syncthreads();
}
c[16*by:16*by+15][tx+64*bx+16*ty] = P1[0:15];
```

(c) Generated CUDA kernel code from script in (b) (N=1024,TI=64,TJ=TK=16)

```
tile_by_index({"i","j"},{TI,TJ},{l1_control="ii",l2_control="jj"},
                    {"ii","jj","i","j"})
tile_by_index({"k"},{TK},{l1_control="kk"},
                    {"ii","jj","kk","i","j","k"},strided)
tile_by_index({"i"},{TK},{l1_control="t",l1_tile="tt"},
                    {"ii","jj","kk","tt","t","j","k"})
tile_by_index({"j"},{TK},{l1_control="s",l1_tile="ss"},
                    {"ii","jj","kk","tt","t","ss","s","k"})
cudaize("mm_GPU",{a=N*N,b=N*N,c=N*N},
              {block={"ii","jj"},thread={"tt","ss"}})
copy_to_shared("tx","b",-16)
copy_to_shared("tx","a",-16)
copy_to_texture("b")
copy_to_texture("a")
copy_to_registers("kk","c")
unroll_to_depth(2)
```

(d) CUDA-CHiLL script for DGEMM on Tesla C2050 Fermi

```
double P3[4][4];
__shared__ double P2[64][17];
__shared__ double P1[16][65];
bx = blockIdx.x, by = blockIdx.y;
tx = threadIdx.x, ty = threadIdx.y;
for (t=0; t<=48; t+=16)
  for (s=0; s<=48; s+=16)
    P3[t/16][s/16] = c[64*by+ty+t][64*bx+tx+s];

for (kk=0; kk<=3952; kk+=16) {
  for (tmp=0; tmp<=3; tmp++) {
    P1[tx][16*tmp+ty] = tex1Dfetch(texbRef,3968*
                  ((16*tmp+64*by+ty) + (kk+tx)));
  }
  __syncthreads();
  for (tmp=0; tmp<=3; tmp++) {
    P2[16*tmp+tx][ty] = tex1Dfetch(texaRef,3968*
                  ((16*tmp+64*bx+tx) + (kk+ty)));
  }
  __syncthreads();
  for (k=0; k<=15; k++) {
  P3[0][0] += P2[tx][k]*P1[k][ty]
  P3[1][0] += P2[tx][k]*P1[k[[16+ty]
  ...
  P3[0][1] += P2[16+tx][k]*P1[k][ty]
  ...
  P3[3][3] += P2[48+tx][tk]*P1[k][48+ty]
  __syncthreads();
  }
  __syncthreads();
}
for (t=0; t<=48; t+=16)
  for (s=0; s<=48; s+=16)
    c[64*by+ty+t][64*bx+tx+s] = P1[t/16][s/16];
```

(e) Generated CUDA kernel code from script in (d) N=3968,TI=TJ=64,TK=16)

Figure 4.1: Compiler generated transformation recipe and automatically generated code for DGEMM.

Chapter 5

Transformation Strategy Generator

This chapter presents the Transformation Strategy Generator (TSG), an automated compiler decision algorithm along with the autotuning mechanism. The algorithms implementing the two contributions of this thesis along with the brief description of the tasks needed as prerequisite for the TSG are presented.

5.1 TSG Overview

The TSG algorithm, as shown in Figure 3.1, proceeds in two phases to derive a set of CUDA-CHiLL transformation commands. The result of the decision making part of TSG is a *master recipe*, listing all the optimization decisions taken. Multiple unique combinations in the form of individual transformation recipes are generated using the master recipe before their evaluation in the autotuning part of TSG.

First, Optimization Strategy Generation generates a *master recipe*, after following two phases:

(I) an analysis of computation partitioning options, which considers dependences, memory coalescing and data reuse;

(II) an analysis of data placement in memory hierarchy, which considers data reuse, sharing patterns and memory coalescing;

The master recipe compactly represents the search space of possible implementations with a collection of multiple computation partitioning decisions as well as all the data

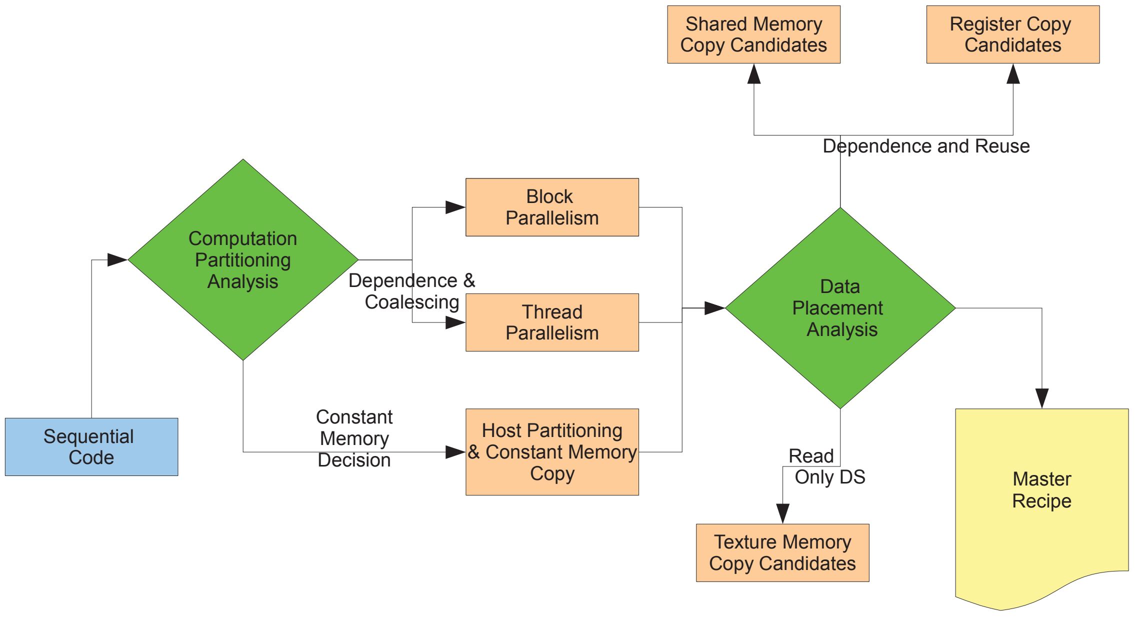

Figure 5.1: Overview of the first phase of TSG, resulting in master recipe.

placement options. The master recipe works as an input to the next phase which spawns multiple solutions by generating unique combinations from its set of code transformation decisions.

Second is the code generation and autotuning part of the system, which consists of two phases:

(III) autotuning a set of unique recipes, generated from combining decisions in the master recipe with default parameter values, and

(IV) autotuning for optimization parameters values for the selected recipe from step (III) and final code variant selection.

We will present each phase in detail.

5.2 Preprocessing of the Loop Order

Before the TSG, we analyze the dependence information in the dependence graph which identifies the loops carrying the dependences. Also in the preprocessing, we gather important information about the data structures in the computation which includes access patterns and dimension. The information about the data structures is needed in reuse analysis, which will determine the movement of data in the memory hierarchy. All the operations to gather the dependence and data structure information are defined in CHiLL.

Once the information is acquired, it is kept, used and modified by the TSG. This allows for the TSG to work independently of the changes being made to the computation related information by underlying software. Due to this independence, the TSG can view, understand and modify the computation at a higher level with a cleaner representation.

5.3 Phase I: Computation Partitioning Analysis

In the first phase of the TSG, the algorithm analyzes the dependence information to partition the computation. The loops in the computation with no dependences will be transformed or tiled, resulting in a new loop nest, where some loops will represent the parallelism (blocks and threads). The computation partitioning analysis algorithm uncovers the possible parallelizations for a given computation, which may be further constrained once memory hierarchy optimizations are applied in Phase II.

Input: The input to the TSG is SUIF intermediate code of the sequential C computation kernel code.

Output: The output of this first phase, as described in the computation partitioning analysis , is a set of computation partitioning variants $v \in V$, each containing candidate loops for blocks (v.BX, v.BY) and threads (v.TX, v.TY and v.TZ) on a GPU, and a loop order (v.Order) that moves these parallel loops to outermost levels, and sequenced such that block loops precede thread loops and according to the x,y or x,y,z dimension sequencing.

The first phase of the algorithm looks at the computation partitioning at three possible levels. These are

- *Kernel-level* partitioning at the host,
- *Block level* partitioning on the device defining a grid of thread blocks, and
- *Thread level* partitioning defining blocks of threads.

5.3.1 Kernel-Level Partitioning at the Host

In CUDA programming model, a computation can be divided at CPU host in to multiple kernels, computing sub-computation. However, the only way one can benefit from such an arrangement, is by the efficient use of memory hierarchy and hiding the latencies associated with the tiling of the computation. In Nvidia GPUs, parts of the memory hierarchy reside in global memory and are efficiently accessed by specific load commands. Example of such memories are texture and constant memory. We know from the literature, that accessing data resident in global memory through constant memory is slightly faster than access through texture or as raw global memory. So we only take the case of using constant memory in the variants with kernel tiling.

We analyze the data structures(typically arrays) used in the computation, to determine if they are read-only and have reuse across thread blocks. If there is any such data structure, it is selected to be assigned for constant memory use. If the size of the data structure is more than the size of constant memory (64K), the kernel has to be tiled in the host code. We create the variant with host level tiling. This adds a *tile_by_index* command, which pushes the tile controlling loop as the outer most. Using constant memory may impact computation partitioning at the device and loop order by the additional level of tiling in the host.

Algorithm: ComputationPartitioningCandidates

Inputs:

```
$L = \langle Loops, Statements, Refs, Deps \rangle$ // a loop nest computation
$v = Initial$ {
    $T = \emptyset$                         // set of tile commands
    $CM = \emptyset$                        // constant memory candidates
    $BX = BY = \emptyset$                   // assignment of block loops
    $TX = TY = TZ = \emptyset$              // assignment of thread loops
    $Order = \{l_1, l_2, \ldots, l_n\}$     // assignment of loop order
    $R = SM = TM = \emptyset$               // candidates for registers, shared and texture memory
}
```

Output: V, master recipe representing a set of partially-specified variants

$Loops = \{l_1, l_2, \ldots, l_n\}$ represents a (possibly imperfect) loop nest in its nesting order

1. *Preprocessing:* Permute *Loops* such that any loop carrying a dependence is moved inward so that the dependence is carried within a thread program, or outward so that it is carried by the sequential host program. If needed, loop skewing can be performed so that the dependences are carried by the outermost loop. The output of this step is a modified loop order, $Order = \{l'_1, l'_2, \ldots, l'_n\}$.

2. *Constant memory candidates:* Identify read-only variables as candidates for placement in constant memory. If a variable's data footprint of the whole loop nest is larger than the constant memory limit, some loops need to be tiled in the host. We create a variant v for each such data structure (one at a time due to a small constant cache) that can be copied together within the same tile, and add the data structures to $v.CM$. The output of this step is a set of loops selected for tiling to adjust the data footprint $v.T$ (potentially empty), and a modified loop order $v.Order$.

3. *Block loop candidates:* Candidates for the block loops must meet the following criteria: (1) they do not carry a dependence; and, (2) it is safe to permute them to the outermost position (or next-to-outermost position if there was a dependence). We use 2D blocks for computations with multi-dimensional data structures, and only a single dimension otherwise (v.BX). This heuristic improves the surface to volume ratio of partitioning. This step (and subsequent ones) is applied to all existing variants, and the output is potentially

a larger set of variants, each incorporating the loops selected for tiling $v.T$, a modified loop order $v.Order$, and the candidate $v.BX$ and $v.BY$ values.

4. *Candidate thread loops in the x dimension and memory coalescing:* Candidate loops for the x thread dimension are evaluated next. Such loops must also meet the following criteria: (1) they do not carry a dependence; (2) permutation to inside the block loops is safe; and, (3) the last dimension of at least one array is accessed by the corresponding loop index. The first is needed for correctness and the third criterion is designed so that the x dimension linearly accesses the bulk of its data in global memory, resulting in coalesced accesses. When there are no loops that meet these criteria, the last one is relaxed. The output of this step is potentially a set of variants, each of which includes the loop selected for tiling $v.T$, a modified loop order $v.Order$, and a candidate $v.TX$ value.

5. *Additional thread dimensions:* Remaining loops mapped to the y and z thread dimensions must meet just two criteria: (1) they do not carry a dependence; and, (2) it is safe to permute them to just within the x thread dimension. As with blocks, we match thread dimensionality to data dimensionality. We may obtain additional thread dimensions by further tiling $v.TX$, as in Figure 4.1(b), or by selecting a different loop for $v.TY$ as in Figure 4.1(d). The output of this step is potentially a set of variants, each of which includes the loops selected for tiling $v.T$, a modified loop order $v.Order$, and candidate $v.TY$ and $v.TZ$ values.

The result is a set of variants which reflect both tiling and no tiling at the host level. The variants with host tiling will have a *tile_by_index* command at this stage.

5.3.2 Block-Level Partitioning

After we have the host level tiling done, we are left with the tiled or untiled loops with dependences and possibly a set of loops without dependences. The loops which are without dependences are selected to setup a maximum of 2 dimension of grid to be assigned to individual SMs for execution. Loops that are candidates for the block loops must meet the following criteria:

(1) they do not carry a dependence; and,

(2) it is safe to permute them to the outermost or next-to-outermost position.

If only one or zero candidate loops carry reuse, we only use a single level of block loops (x dimension). Otherwise we parallelize maximum allowed two loops for x and y dimensions. This comes from the heuristics of improving surface to volume ratio of partitioning. Once candidates for the x and y dimensions are selected, we mark these loops for tiling, so that tiling can be applied to adjust the number of blocks during autotuning. The output of this step is potentially a set of variants, each incorporating the loops selected for tiling , a modified loop order , and the candidate values for the tile sizes which ultimately decide the number and dimension of blocks in the grid.

5.3.3 Thread level partitioning

Candidate loops for the x-thread-dimension are evaluated next. Such loops are the ones that are not marked as block loops and must also meet the following criteria:

(1) they do not carry a dependence;

(2) permutation to just inside the block loops is safe; and,

(3) the last dimension of at least one array is accessed by the corresponding loop index.

The first criterion is needed for correctness and the third criterion is designed so that the x-dimension linearly accesses the bulk of its data in global memory, resulting in coalesced accesses. When there are no loops that meets these criteria, the last one is relaxed.

5.4 Phase II: Data Placement Analysis

The second phase examines each data structure in the computation and identifies candidates for mapping to different levels of the memory hierarchy (other than constant memory). The data placement analysis algorithm takes as input the master recipe from Phase I with parallelization decisions. The information about the data structures collected in pre-processing is used to identify the access and reuse patterns along with the structure of the arrays. Data placement decisions are made for each possible parallelization strategy. It marks which data structures can be destined for which level of the memory hierarchy. The algorithm explains the criteria for data placement in shared memory, registers and texture memory. While placement into texture memory is constrained to read-only data, the decisions of placement in shared memory and registers are based on reuse across or within the threads.

Algorithm: DataPlacementCandidates

Inputs:

$L = \langle Loops, Statements, Refs, Deps \rangle$//a loop nest computation

V = master recipe from Phase I with parallelization candidates

Output: V, master recipe with parallelization and data placement candidates

1. *Shared memory candidates:* Candidates for shared memory $v.SM$ are not in $v.CM$. They have reuse across one of the thread loops $v.TX$, $v.TY$ or $v.TZ$, or their accesses are not coalesced in $v.TX$ and coalescing is possible for other loops $l \in Loops - \{v.BX, v.BY, v.TX\}$. For each candidate data structure d, the algorithm creates a shared memory candidate object $sc = \langle d, l, t, ct, o \rangle$ that describes the data structure, a loop level l for which the copy should be placed, and, optionally, a specification of how to tile internal loops within each thread. This specification includes a set of internal loops t to be tiled, corresponding tile controlling loops ct, and a modified loop order o after tiling. These latter three optional fields are only necessary if d's footprint within l is an array that is larger than the number of threads, as determined by polyhedral scanning. For example, the loop order o is defined as $\{v.BX, v.BY, ct, v.TX, v.TY, v.TZ, t\}$ if t represents a single loop. This order captures the right footprint for the copy across threads. We add each sc to $v.SM$.

2. *Register candidates:* Candidates for register $v.R$ are not in $v.CM$ or $v.SM$, and have reuse carried within the innermost loop or across some other loop inside the thread program. For each candidate data structure d, the algorithm creates a register candidate object $rc = \langle d, l, t, ct, o \rangle$ that describes the data structure, a loop l to place the copy, and an optional

tiling specification as for shared memory tiling. For registers, the loop order o when tiling for a single loop t within the thread would be $\{v.BX, v.BY, v.TX, v.TY, v.TZ, ct, t\}$. We add each rc to $v.R$.

3. *Texture memory candidates:* Candidates for texture memory are read-only data that are also in $v.SM$. We have found through experimentation that texture memory has only demonstrated performance gains in such cases (and not for data in registers or global memory).

5.5 Phase III: Generating and Autotuning Variants

At this stage, the TSG has finished laying out the optimization decisions for computation partitioning and data placements. These decisions, collected in the *master recipe*, serve as the input to this phase of code variant generation and evaluation. The autotune-variant algorithm will generate actual CUDA-CHiLL recipes from all unique combinations of decisions prescribed in the master recipe. It shows how the algorithm combines and emits the commands representing the individual optimization decisions in the master recipe. It has to make sure that commands which are paired with other commands are emitted correctly in a given recipe. For example, a certain data placement command may only work if it is accompanied by a *tile_by_index* command, tiling the copy of data in a certain part of memory hierarchy. These commands also have some parameters which can have multiple values. The potential combinations of the all the values of the parameters with the set of unique combinations of optimizations can be huge. Thus we adopt a strategy to split the evaluation in to two phases. A preliminary pruning is done at this phase by binding a fixed value to each of the parameters in the recipe and empirically evaluating the recipes, to pick the best in the set. The output is often a single recipe which performs the best, with parameter values which might not be optimum.

We validated this pruning strategy by doing an exhaustive search of parameters and data placement for matrix-matrix multiply and found that, while the actual performance varies with different parameter values, the relative differences between implementations

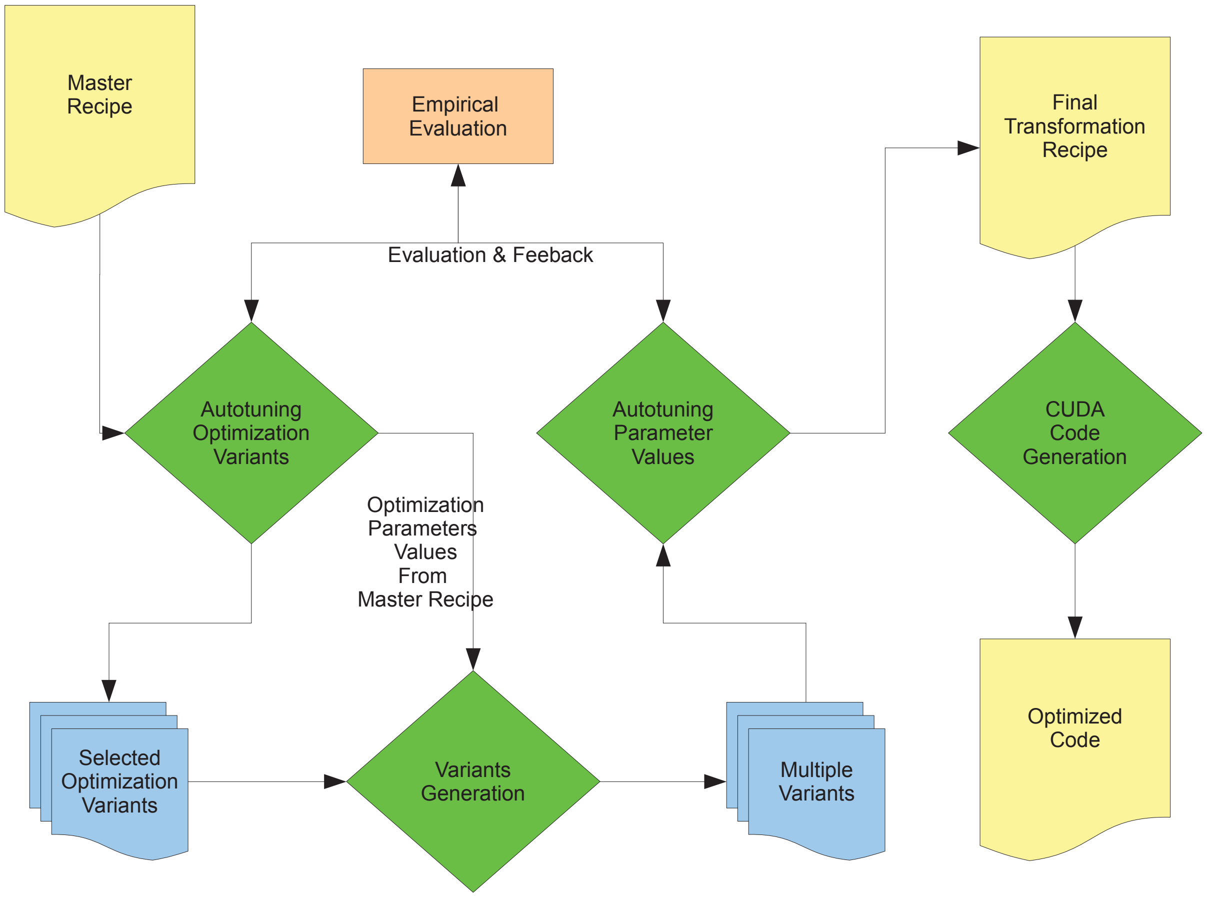
Master Recipe
Empirical Evaluation
Final Transformation Recipe
Evaluation & Feeback
Autotuning Optimization Variants
Autotuning Parameter Values
CUDA Code Generation
Optimization Parameters Values From Master Recipe
Selected Optimization Variants
Variants Generation
Multiple Variants
Optimized Code

Figure 5.2: Overview of the autotuning and empirical evaluation phases of the TSG.

leads us to the right data placement decisions. Therefore, for this benchmark which is coming closest to peak performance, we can prove that this approach to separating the data placement search from the parameter search does not degrade the final solution while dramatically reducing the number of points to be searched.

Algorithm: AutotuneVariants

Input: V = master recipe from Phase II

Output: V' = set of parameterized CUDA-CHiLL scripts

1. *Constant memory:* If this variant uses constant memory, the algorithm emits the following for all data structures d in $v.CM$: *emit(v,copy_to_constant(d))*

2. *Reorder and tile for computation partitioning:* Emit the command to tile and order block and thread loops. If this variant requires tiling in the host for constant memory, the algorithm also emits a tile command to accomplish this. Prior to generating the command, the algorithm creates parameters representing tile sizes $T_1, \ldots T_n$ and identifiers representing tile controlling loops $t_1^c, \ldots, t_n^c$. This step generates the ordering for the block and thread loops, and applies the necessary tiling to adjust parallelism as needed: *emit(v,tile_by_index(*$v.T$*,* $\{T_1, \ldots, T_n\}$*,* $\{l1_{control} = l_{t_1^c}, \ldots\}$*,* $v.Order$*))*

3. *Copy to registers, tiling if needed:* The algorithm creates a copy v' of each variant $v \in V$ for each subset $v'.R \subset v.R$, and adds it to V'. This allows TSG to evaluate which subset yields the best utilization of the register file, as follows:

   (a) Scan the set $v'.R$ to identify whether additional tiling is needed from the individual $rc = \langle d, l, t, ct, o \rangle$ objects in $v'.R$.

   (b) For example, assume only a single loop t must be tiled, the tile size T is created, and the remainder is specified in the register candidate objects.

   (c) Generate tile commands as needed: *emit(v',tile_by_index(t,* $\{T\}$*,* $\{ct\}$*, o, strided))*

   (d) Foreach $rc \in v'.R$, *emit(v',copy_to_registers(rc.d,rc.l))*

4. *Copy to shared memory, tiling if needed:* The algorithm creates a copy v' of each variant $v \in V$ for each subset $v'.SM \subset v.SM$ and adds it to V'. This allows TSG to evaluate which subset yields the best utilization of the shared memory, as follows:

   (a) Scan the set $v'.SM$ to identify whether additional tiling is needed from the individual $sc = \langle d, l, t, ct, o \rangle$ objects in $v'SM$.

   (b) For example, assume only a single loop t must be tiled, the tile size T is created, and the remainder is specified in the shared memory candidate objects.

   (c) Generate tile commands as needed: *emit(v',tile_by_index(t,* $\{T\}$*,* $\{ct\}$*, o, strided))*

   (d) Foreach $sc \in v'.SM$, *emit(v',copy_to_shared(sc.d,sc.l))*

5. *Fetch using texture memory:* The algorithm creates a copy v' of each variant $v \in V$ for each subset $v'.TM \subset v.TM$ and adds it to V'. This allows TSG to evaluate which subset yields the best utilization of texture memory, as follows:

   (a) Foreach $tc \in v'.TM$, *emit(v',copy_to_texture(tc.d))*

6. *CUDA blocks, threads and kernel:* Generate the CUDA command for each variant $v \in V$: *emit(v, cudaize(⟨ kernel ⟩, { ⟨params⟩ }, { block={$v.BX, v.BY$}, thread={$v.TX, v.TY, v.TZ$ } }))*

7. *Loop Unrolling and Unroll-and-Jam:*

   As a final step, the algorithm creates up to three variants $v1, v2, v3$ for each variant $v \in V$ corresponding to different levels of unrolling: *emit(v1, unroll_to_level(0))*, *emit(v2, unroll_to_level(1))*, *emit(v3, unroll_to_level(2))*

5.6 Phase IV: Autotuning Optimization Parameter Values

From Phase III, a single recipe is selected and serves as input to the last phase of the TSG. The focus of the Phase III was to decide on the unique combination of optimizations from a set, with bounded parameter values. The focus of the Phase IV is to allow for multiple values for parameters in the recipe selected from Phase III. The autotuning parameterized variants algorithm describes the final autotuning step on unbound parameters in the recipes to find the best parameter values and the best recipe.

Algorithm: SelectBlockTileSizes

X=$MAX\_NUM\_OF\_THREADS\_PER\_BLOCK$

$T = \{warp\_size, warp\_size * 2, ...X\ \}$

M = number of Streaming Multiprocessors

Input:

```
        $P$ = problem size
```

Output: T' ={empty set }// tile sizes potentially maximizing device occupancy.

```
        foreach ($t \in T$)
          # of blocks (B) = ceil($P$/t)
          # of blocks in last computation cycle (R) = (B%M)
          If ( M-1)/2 < R < (M-1)
            $T' \leftarrow$ t
```

Algorithm: Autotuning Parameterized Variants

P = set of problem sizes

Input:

```
        $Tl = \{pr_1, pr_2....pr_n\}$
        $Tl\_comb$ = {empty set}
        $V$ = selected recipe from Phase III
        $t\_val$ = MAX_VAL
```

Output: V' = optimized code

```
  foreach ($p' \in P$)
    foreach ($i \in Tl$)
      $Tl[i] \leftarrow$ SelectBlockTileSizes(p')
      $Tl\_comb = \{Tl[1] \times\ Tl[2] \times\ .... \times\ Tl[n]\ \}$
      $v = V$
      foreach ($t \in Tl\_comb$)
        $v.t \leftarrow$ t
        val = evaluate(generate_code($v$))
        If val < t_val
          $V' \leftarrow$ generate_code($v$)
```

5.7 Selecting Parameter Values: Tile Sizes

In optimizing GPU code through compiler transformations and optimization techniques, we see the problem as essentially a tiling problem. However, the fact that we have to autotune through the parameters for tiling, presents us with a challenge to constrain the search space for its parameters to a small subset of values. Also important is the fact that, to generate the optimized libraries for a set of functions we need to provide a solution based on some limited set of values (2 or 3) for tiling parameters, where the performance difference is not more than 5 or 10% of the best solution. We focused our study on optimizing CUDA code for NVIDIA GPU (GTX 480). We started looking at an aspect of GPU code execution which is stressed upon by a number of researchers and library developers i.e. occupancy. Occupancy here refers to the use of available GPU resources, basically how much work is scheduled for SMs in the course of complete execution of a piece of code. Many contemporary researchers believe higher occupancy is necessary to achieve greater performance [Nvi08b, YXKZ10]. Others like Vasily et al. [VD08b] argue against occupying much of the GPU in terms of blocks, but by using as much of the fastest memory available as possible. It seems that almost comparable performances can be achieved by both the methods with appropriate parameter values for different variables like block sizes, thread workload, number of registers and amount of shared memory usage etc. In this discussion we will focus on the question of whether we can predict with enough generalization, the set of tiling parameter values that may always lead to a solution with comparable performance to the best available.

5.7.1 Understanding Features and Constraints

Consider matrix-vector multiplication as an example. Our optimization strategy for this BLAS kernel is based on tiling the data for shared memory and registers and coalescing accesses to global memory from within the threads. We have a 1D block and thread dimension within a block organization for the distribution of work on the streaming multiprocessors of the GPUs. To start with, we concentrated on two tile sizes i.e. 16 and 32 which correspond to the half-warp and warp sizes (the unit of resource allocation) in CUDA. In our strategy, the tile sizes are also important, in the context of determining the number of threads in a block and thus the block size. Also effected would be the number of registers and amount of shared memory use, effected by another parameter, which is the amount of work to be delegated to each thread in a block. We arrange the threads in a single row, for coalescing global memory accesses and assign a vertical stream of work for each thread, accomplished by having a loop inside each kernel implementation intended for threads. For this particular kernel, the loop size is also the same as the tile size. So in this particular case, one tile size is effectively dictating all the important occupancy, memory utilization and work management issues. The example of matrix-vector multiplication can be considered a base case for understanding the relation of particular tile sizes with the available system resources. So we devised a set of experiments to understand and to come up with a generalized formula for selecting tile size, which performs better based on the notion of effective use of available resources.

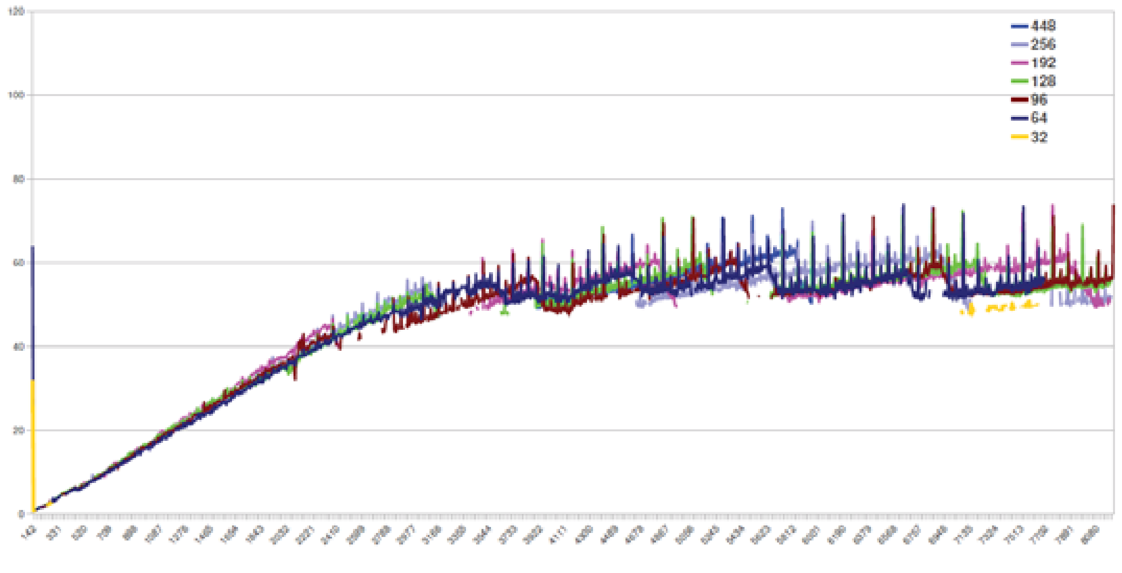

Figure 5.3: Experimental evaluation of the most profitable tile sizes for matrix-vector multiplication over a range of matrix problem sizes (128-8K).

5.7.2 Efficient Workload Management: Experimental Methodology

We performed separate experiments with different tile sizes for a range of problem sizes from 128-8k, with step size of 7, and identified the best tile size at each point. The tile sizes we focused on eventually, were multiples of 32, the warpsize.

The hypothesis was that if we could find a pattern in the behavior exhibited by each tile size in relation to the problem size, on a particular set of resources, we can at least constrain the possible tile sizes in a certain range, which would then enable us to pick two or three best ones with not a lot of performance difference among them. The hypothesis was based on the experimental observations, where we manually tried a particular set of tile sizes (multiples of 16 or 32), striving for better performance. We found out that some tile sizes performed poorly compared to the other tile sizes, on the same problem sizes. Periodic drops in performance were clearly observed in the curves of each tile size as shown in Figure 5.3.

We observed a drop in performance for numbers of blocks that were not a multiple of 16. This behavior can be attributed to under utilization of the available SMs by the number of blocks remaining in the last batch of executing blocks. We observed that every time the utilization of SMs was <50% at a given time, the performance would fall, resulting in pruning of the particular tile size.

An interesting fact evident from the graph is that the periodicity of performance drop is inversely proportional to the size of the tile. The effects are more visible for larger problem sizes ($> 2K$). For $< 2K$, we can almost pick any of the multiples of 32. From these observations, we have developed an algorithm to select tile size values.

5.7.3 Selecting Tile Sizes to Maximize Device Occupancy

The algorithm to select block tile sizes outlines the process of selecting tile size values. This simple method gives us the set of tile sizes values which can then be autotuned to find the best one. Manually-tuned libraries use a set of implementations for a range of problem sizes. We can generate these implementations automatically through the autotuning of the parameter values. A set of two to three parameter values can be used to generate a set of implementations of a computation for a given range of problem sizes.

Tile Sizes: Threads (x dimension) The scheduling unit on a GPU is a *warp* of 32 threads. Of particular importance are the number of threads in the x dimension, which exploit global memory coalescing. Thus the algorithm limit the tile size used to govern number of threads per block by the warp size and its multiples. Also, by using tile sizes to define the number of threads in a block, we put an upper bound on the tile size search space, i.e. maximum number of threads per block.

Tile Sizes: Blocks The algorithm takes into account a data set size (or a range) to determine whether a particular tile size leads to a computation partition where a significant fraction of the resources are idle. This algorithm was derived from empirical analysis using a much larger set of tile size candidates, and prunes away sizes that will lead to idle resources.

Tile Sizes: Data Staging When tiling is used to stage data into shared memory or registers, the maximum tile size is limited by shared memory or register capacity constraints, respectively. The footprint of data accessed by a thread block cannot exceed the size of shared memory, which is quite small as shown in Table 2.1.

5.8 Summary

This chapter presented the details about the TSG and autotuning mechanisms, two of the major contributions of this thesis. The chapter presented the key constraints and heuristics driving the decision making. Also presented and discussed in detail are the algorithms defining the functionality of these parts in the system.

Chapter 6

Experimental Evaluation

We generate highly-tuned code for the benchmarks presented in Table 2.2 on the two architectures studied for this thesis i.e. the GTX-280 and C2050 presented in Table 2.1.

6.1 Experimental Methodology

For all measurements, each version of the generated code and the CUBLAS library was evaluated 10 times, and we used the mean of those run times, recorded using CUDA events. In the case of benchmarks from a contemporary compiler [BRS10], we have used the timing mechanism provided with their benchmark suite which is based on C timers. Performance results were reasonably stable, with a standard deviation of less than 0.1msec for execution times in the tens of milliseconds. Prior to showing all the performance results, we describe in detail two case studies and compare against manual tuning.

6.2 Performance Comparison: Manually-Tuned BLAS

This section presents the results of optimized code for three BLAS kernel routines compared with those in the manually-tuned BLAS library by Nvidia, CUBLAS.

6.2.1 Matrix-Matrix Multiplication

Matrix-matrix multiplication, from the set of BLAS kernels, was discussed as a case study in Chapter 4. Figures 6.1 and 6.3 show the performance of SGEMM and DGEMM on the two GPU architectures over a range of square matrix sizes from 1K to 8K elements. We compare against the CUBLAS 3.2 library, and for C2050 we also compare against

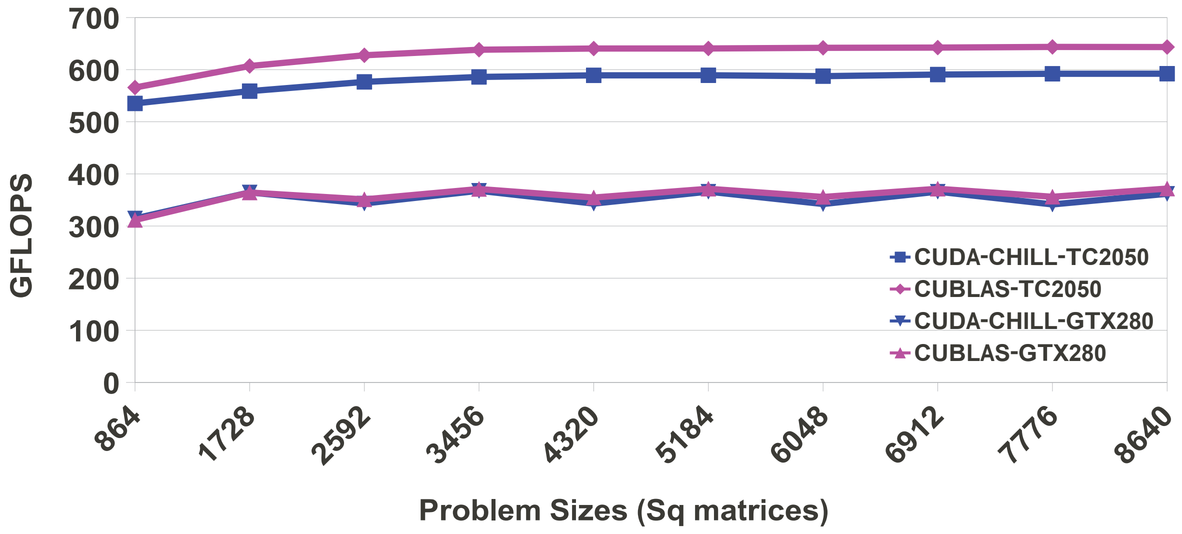

Figure 6.1: SGEMM GFLOPS (vs CUBLAS 3.2).

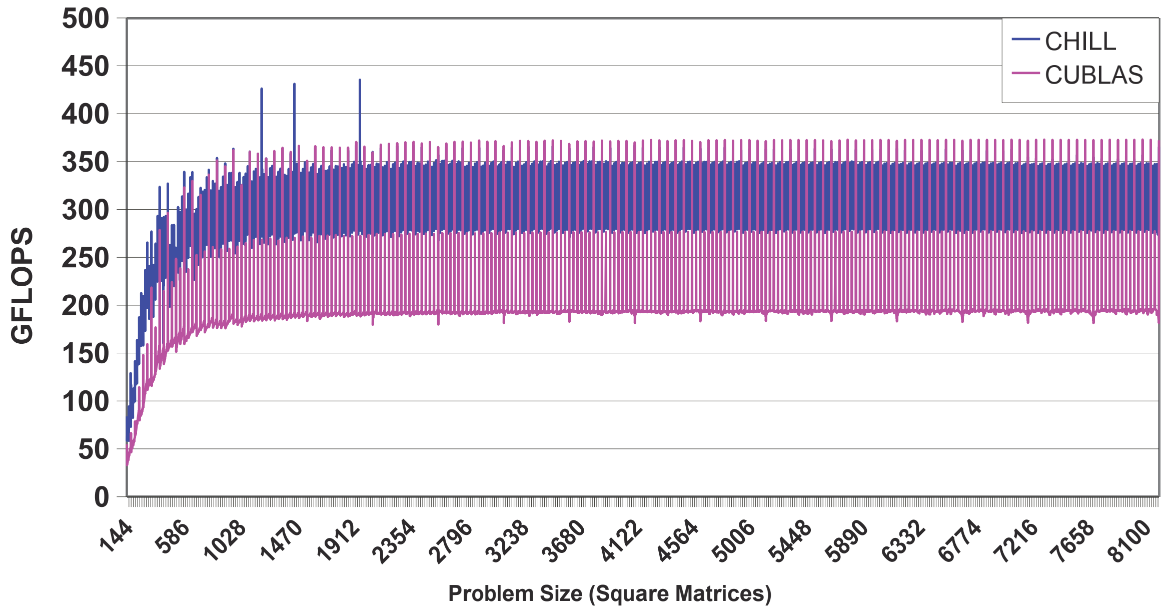

Figure 6.2: SGEMM GFLOPS (vs CUBLAS 2.2 for GTX-280).

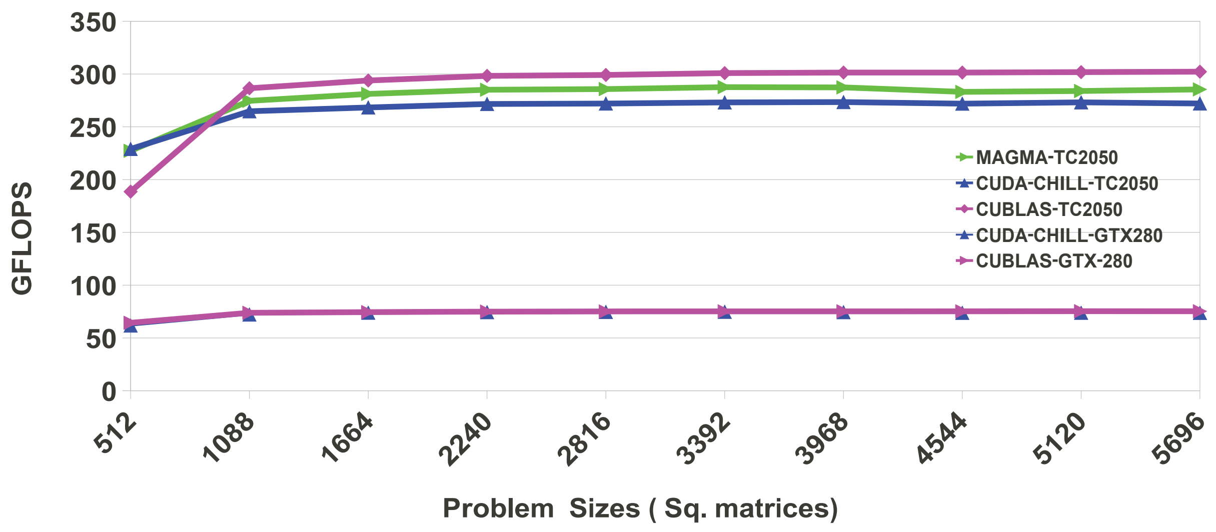

Figure 6.3: DGEMM GFLOPS (vs CUBLAS 3.2).

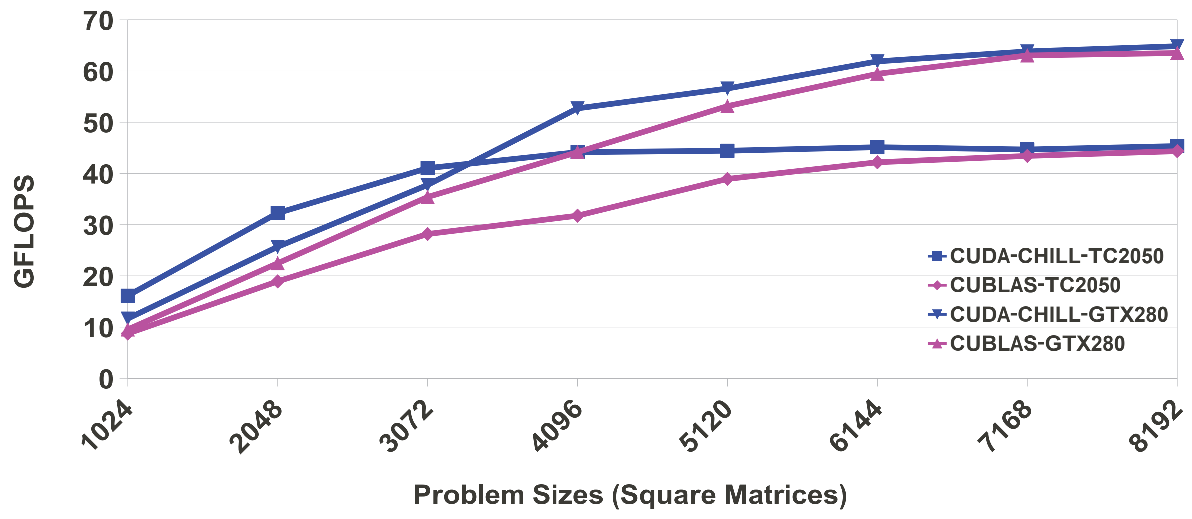

Figure 6.4: SGEMV GFLOPS (vs CUBLAS 3.2).

MAGMA 0.2 [NTD10] for DGEMM. Figure 6.1 shows SGEMM, which achieves the highest level of performance (up to 592 GFlops) for the compiler-generated code because it has the most computation. The compiler-generated code performs almost at the same

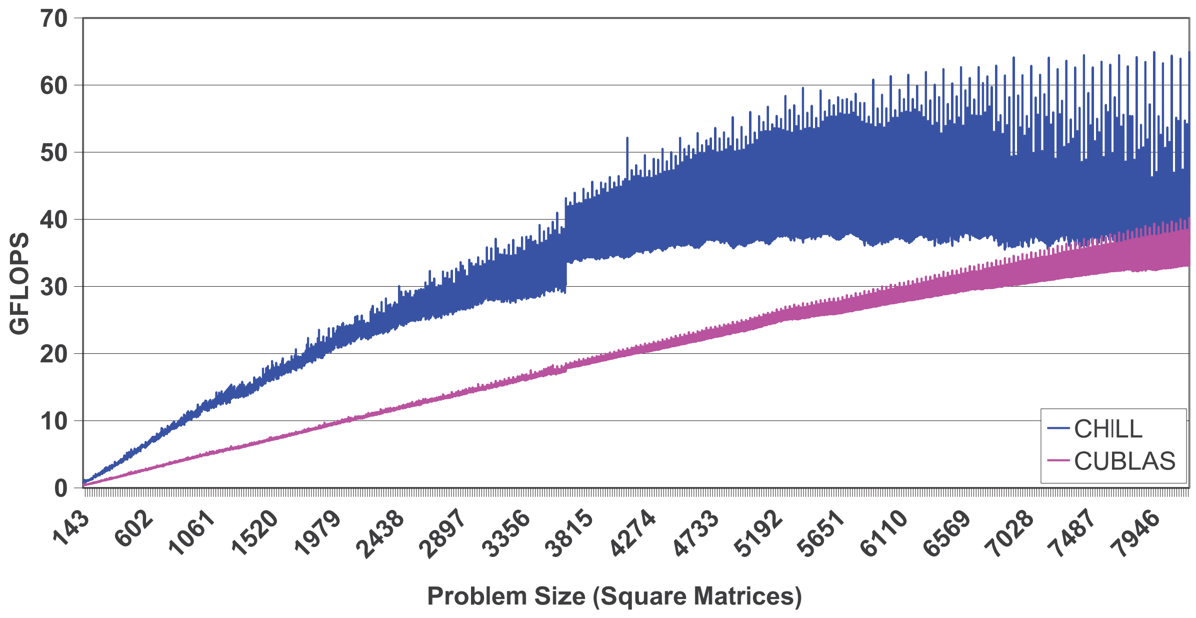

Figure 6.5: SGEMV GFLOPS (vs CUBLAS 2.2 for GTX-280).

level as the manually-tuned CUBLAS 3.2, coming within 2% of CUBLAS 3.2 on the GTX-280 and within 8% on the Tesla C2050. MAGMA performance lies in between the two, but all are close.

We have also compared the code against a previous version of CUBLAS i.e. CUBLAS 2.2 on GTX-280, as shown in Figure 6.2. The experiment was done on every point from problem size 128 up to 8K. The graph shows the optimized code from our system performing consistently better than CUBLAS 2.2. There are the problem sizes in the range, where CUBLAS performs comparable or better than code from our compiler with manually-tuned routine. The same routine, if used for other points throughout the range, does not yield the same kind of performance. It strengthens our conviction that generating customized solutions for computation with slightly different parameters, yields superior performance compared to the use of fixed implementations.

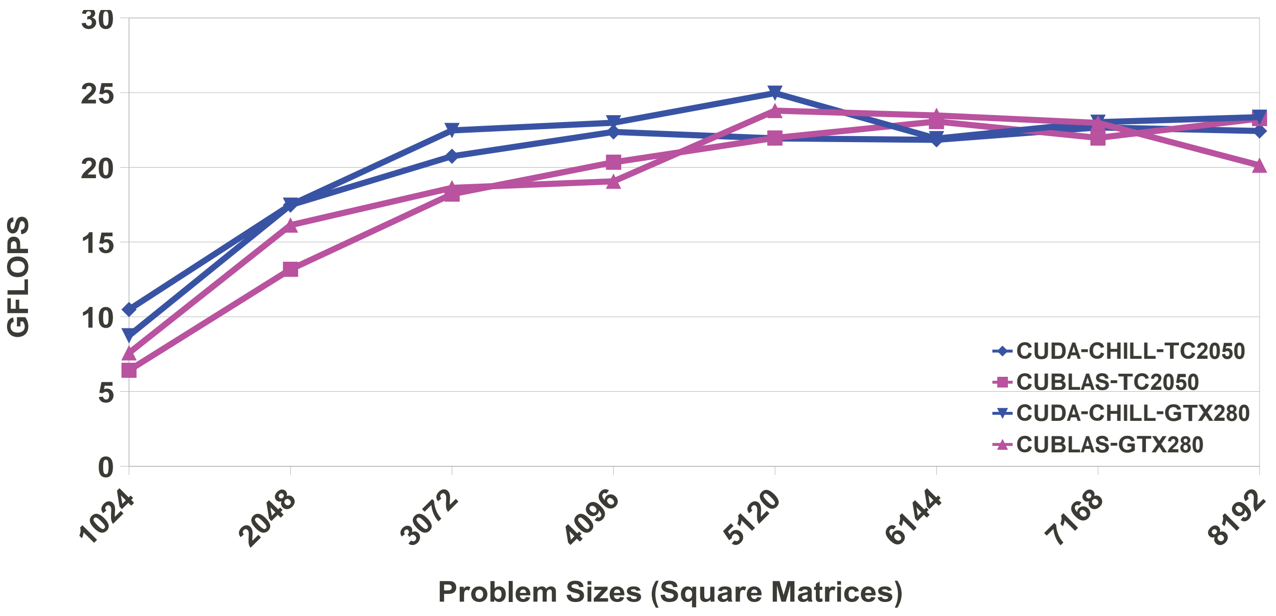

Figure 6.6: DGEMV GFLOPS (vs CUBLAS 3.2).

6.2.2 Matrix-Vector and Transposed Matrix-Vector Multiplication

We will now discuss the results of the other BLAS benchmarks, which include single and double precision results for matrix-vector multiplication (GEMV), and transposed matrix-vector multiplication (TMV), and compare against CUBLAS. The matrix-vector multiplication kernels is optimized by choosing a single dimension of blocks and threads for computation partitioning. The input vector is put in shared memory and the result is stored in registers by each thread. Much of the performance improvement is done through coalesced accesses loading the entire vector into shared memory for most problem sizes.

As compared to GEMV, the challenge in achieving high performance for transposed matrix-vector multiplication is that it is not possible to achieve global memory coalescing for the input matrix given the parallelization strategy, where each output element is assigned its own thread. Therefore, the compiler loads the input matrix into shared memory in coalesced order, and then different threads access the data.

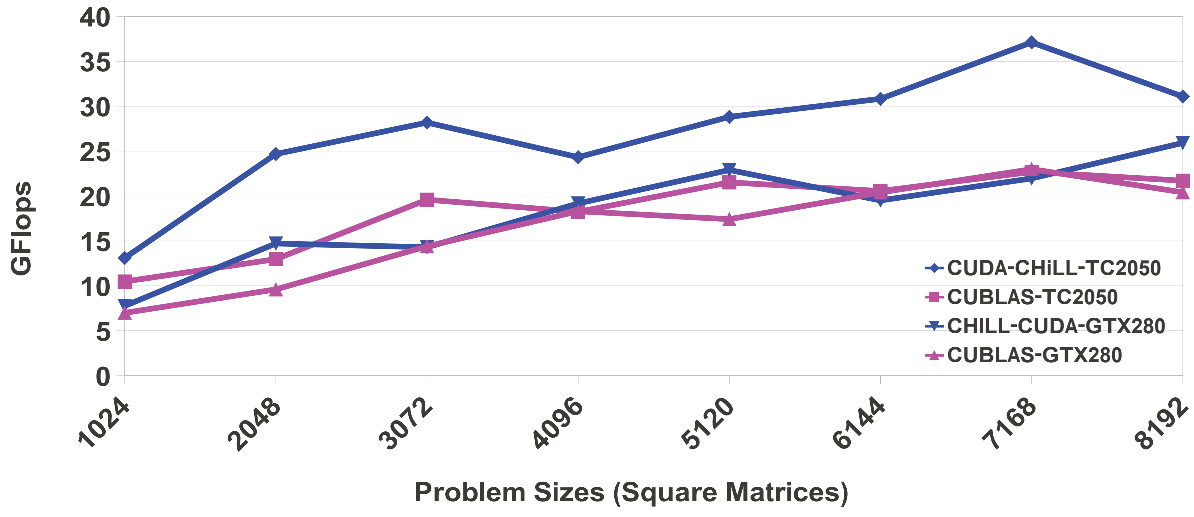

Figure 6.7: SGETMV GFLOPS (vs CUBLAS 3.2).

We use constant memory for the input vector, and it improves performance for all but one problem size (8K) on GTX-280 and for a single problem size (6K) on C2050, in our single precision experiments. Double precision results also follow the same trend, but times out on the GTX-280 for larger problem sizes. On GTX-280, which has no L1 cache and smaller shared memory, single precision code always perform within 5% of CUBLAS, with a maximum speedup of 1.53x. Our double precision code achieves 1.90x speedup over CUBLAS. For C2050, the code outperforms CUBLAS significantly, up to 1.90x for both single and double precision code.

For SGEMV in Figure 6.4 the compiler-generated code outperforms CUBLAS 3.2 on both architectures, by as much as 1.84x on C2050 and 1.22x on GTX-280. It is interesting that the raw performance for larger data set sizes is actually better on the less powerful GTX-280. We suspect this reflects a less efficient use of the available memory bandwidth on the C2050. The compiler uses a register for the result, shared memory for the input vector, and accessing the input array out of global memory, but tile sizes are different for the two platforms, generally smaller sizes for GTX-280 (64-128 range) and larger for C2050 (128-384 range). Results for TMV are shown in Figure 6.7 and 6.8.

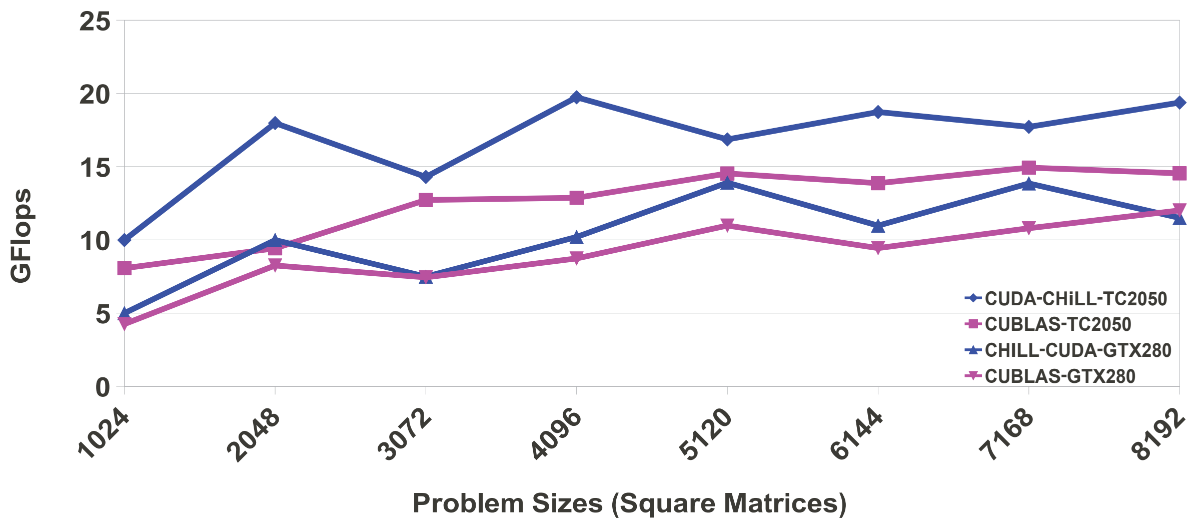

Figure 6.8: DGETMV GFLOPS (vs CUBLAS 3.2).

Figure 6.5 presents comparison against CUBLAS 2.2 for GTX-280. The automatically generated code from our compiler system almost always performs better than CUBLAS 2.2 on each point from 128-8K. The automatically-generated code for both benchmarks almost always outperform the manually-tuned CUBLAS code.

6.3 Performance Comparison with State-of-the-Art GPU Compiler

We applied our system to a set of optimized benchmarks generated by a state-of-the-art compiler presented in [BRS10]. To derive timings, compute GFlops and compare with the benchmarks, we used the test harness provided with the benchmarks. Figure 6.9 shows the speedups we achieve over the [BRS10] benchmarks (y-axis), on the C2050 and GTX-280 GPUs; each bar represents a benchmark. Performance in GFlops for each kernel is shown at the top of each bar. Overall, our system achieves an average speedup of 1.48X and 1.33X for the C2050 and GTX-280, respectively.

The CP benchmark shows the highest performance (332 GFlops) and speedup (2.03X) over PLUTO. Our system selects 32x16 blocks and 16x16 threads and fully unrolls the

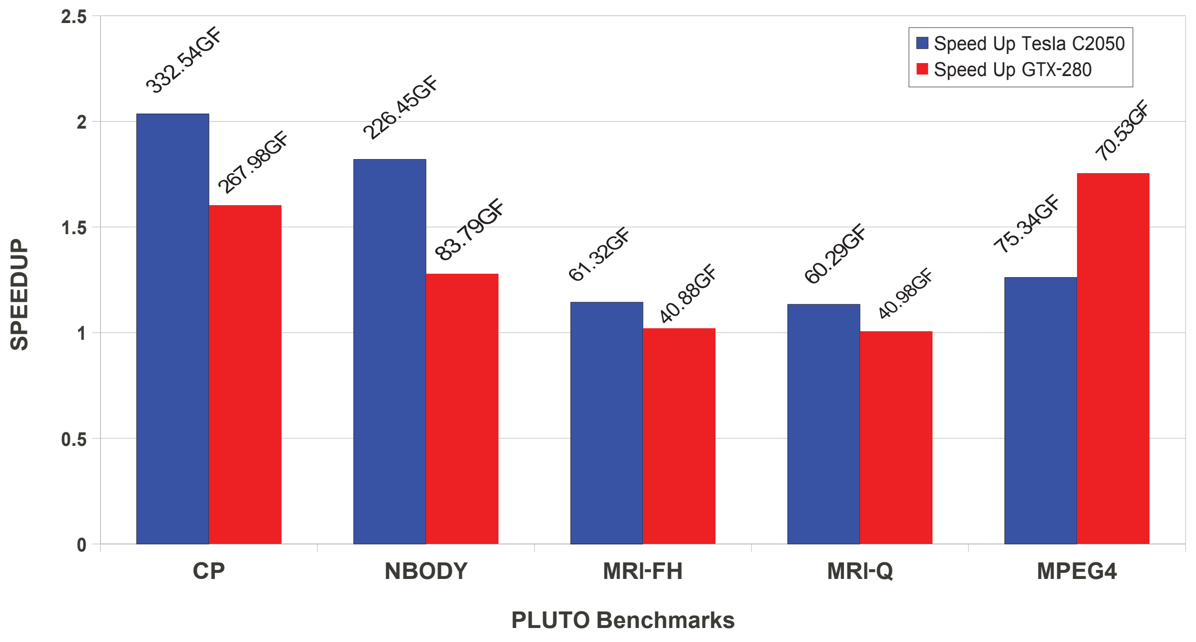

Figure 6.9: Performance comparison with state-of-the-art GPU compiler in [BRS10]

innermost computation loop of 256 iterations, while PLUTO uses 32x32 blocks, 16x16 threads and less work per thread, along with an unroll factor of 16 (using an unroll pragma). A similar strategy achieves performance of over 265 GFlops on the GTX-280.

For the two MRI kernels, MRIQ and MRI-FH, we were able to find better performing solutions by using half as many threads per block as the PLUTO-generated code. In addition, in MRIQ, data placement in shared memory for the key data structure $kVals$ was also different from the PLUTO-generated code. The code for MRIQ is also highlighted in Chapter 8 of [KH10].In [KH10] and [IMP07], a different strategy is used for $kVals$ that tiles in the host code and places tiles of $kVals$ in constant memory, as is done in the PLUTO benchmark. The PLUTO and hand-coded PARBOIL benchmarks MRIQ and MRI-FH perform comparably to each other; and our compiler achieves a speedup of 1.13X on the C2050 and matches their performance on the GTX-280.

In the NBody kernel, the major differences from the PLUTO strategy are fewer threads per block and unrolling of the innermost computation loop with a factor of 128

```
  struct kValues {
   float Kx; float Ky; float Kz; float PhiMag;
  } kVals[M];
  float x[N], y[N], z[N], Qr[N], Qi[N];
  for ( i = 0; i < M; i++) {
    for ( j = 0; j < N; j++) {
      expArg = PIx2 * (kVals[i].Kx*x[j] + kVals[i].Ky*y[j] + kVals[i].Kz*z[j]);
      cosArg = cosf(expArg);
      sinArg = sinf(expArg);
      phi = kVals[i].PhiMag;
      Qr[j] += phi * cosArg;
      Qi[j] += phi * sinArg;
    }
}
```

```
permute(0,{"j","i"})
tile_by_index({"j"},{TI},{l1_control="jj"},{"jj","j","i"})
normalize_index("j")
normalize_index("i")
tile_by_index({"i"},{TJ},{l1_control="ii",l1_tile="i"},
                    {"jj","ii","j","i"})
cudaize("kernel_GPU",{x=N,y=N,z=N,Qr=N,Qi=N,kVals=M},
              {block={"jj"},thread={"j"}})
copy_to_shared("tx","kVals",1)
copy_to_registers("ii","x")
copy_to_registers("ii","y")
copy_to_registers("ii","z")
copy_to_registers("ii","Qi")
copy_to_registers("ii","Qr")
```

Figure 6.10: MRIQ: Sequential code and transformation strategy for C2050.

compared to PLUTO's factor of 16 unrolling. Our system achieves a 1.81X and 1.28X improvement on the C2050 and GTX-280, respectively. For MPEG4, a multimedia kernel, the computation partitioning and data placement strategy is similar across our compiler and PLUTO, but we use 256x256 blocks as compared to 8x4 blocks in PLUTO. The number of threads per block is the same, but our solution performs far less work per thread, and allows the system to co-schedule multiple blocks on the same multiprocessor. We achieve a 1.26x speedup on C2050 and 1.77x on GTX-280 as compared to PLUTO.

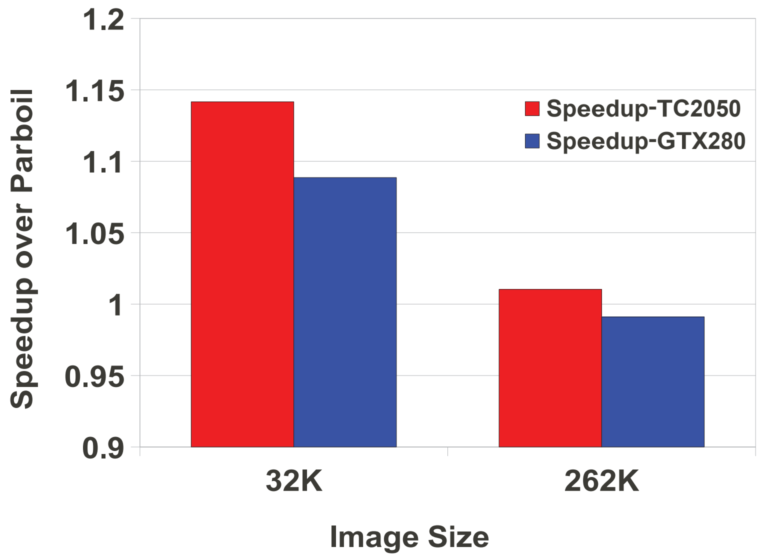

Figure 6.11: MRIQ GFLOPS.

6.4 Performance Comparison of MRIQ and 2D Convolution

We briefly consider the benchmark MRIQ. from Parboil [IMP07] and PLUTO [BRS10] benchmark suites. This code is highlighted in Chapter 8 of [KH10].The automatically-generated script for both architectures places a key data structure $kVals$ in shared

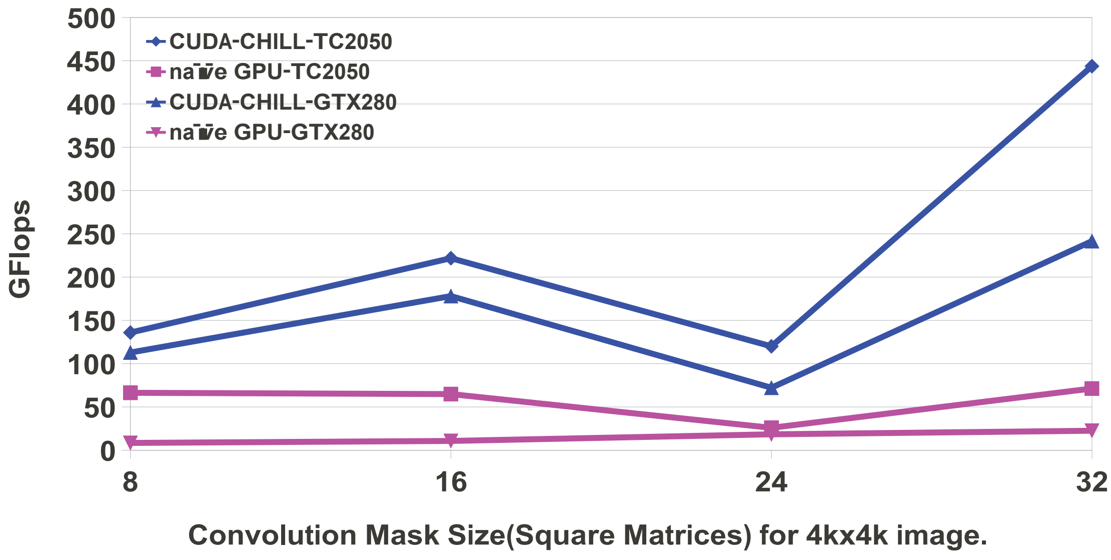

Figure 6.12: 2D Convolution GFLOPS.

memory and all other data in registers. In [KH10] and [IMP07], a different strategy is used that tiles in the host for constant memory.

CUDA-CHiLL achieves speedups over the manually-tuned implementation for large data sets of 1.14x on the Tesla and 1.08x on the GTX-280, as shown in Figure 6.11.

2D Convolution. To show that this approach can be applied to optimize other array-based codes, the last benchmark is a 2D convolution, applied to a square image of 4K but with varying mask size and an irregular data footprint for the image. We applied our optimization for four different mask sizes, from 8 to 32. Changing the mask sizes results in different tile sizes. Thus the problem size defines the computational decomposition to achieve the best performance. We found the use of constant memory for the mask matrices and shared memory for the image itself, to be the most beneficial.

The automatically-generated code achieves up to a 13x speedup over the naive implementation, and up to 1.67x improvement on the compiler described in [YXKZ10] as shown in Figure 6.12. We achieve performance ranging from 72 GFlops to nearly 250 Gflops on GTX 280, and up to 443Gflops on Tesla C2050 with a minimum of over 120Gflops, depending on the mask size (which affects amount of data reuse in shared memory).

6.5 Impact of Optimizations on Performance

In this section we show the impact of specific optimizations on application performance. The code targeting the two architectures have a few similarities and use all the various levels of the memory hierarchy. Nevertheless, the impact of the optimizations vary across architectures, applications and data set sizes. In Table 6.1, we capture the performance difference from the best-performing baseline, and one with one optimization turned off.

Overall, we observe that the most significant loss of performance results from either disabling unroll-and-jam or copying to registers.These optimizations work together to

	Problem	Unroll		Registers		Shared Mem		Texture Mem		Constant Mem	
Name	Size (Sq. Mat)	GTX-280	Tesla C2050	GTX-280	Tesla C2050	GTX-280	Tesla C2050	GTX-280	Tesla C2050	GTX-280	Tesla C2050
SGEMM	4320	92.30	97.12	92.34	84.77	72.17	97.68	1.38	4.20	0.00	0.00
DGEMM	4544	Timedout	95.39	Timedout	94.55	37.60	53.20	0.52	3.28	0.00	0.00
SGEMV	4096	67.78	63.46	67.31	70.42	54.10	0.00	0.00	0.00	0.00	1.31
DGEMV	2048	48.93	56.75	57.89	61.91	0.00	0.00	0.00	0.00	2.69	3.34
SGETMV	4096	63.06	0.51	29.71	20.75	92.59	77.32	0.00	0.00	3.84	0.00
DGETMV	2048	0.00	0.00	9.67	19.54	85.26	83.33	0.00	0.00	0.00	0.00
2D-Conv (single precision)	4k,16	58.66	64.73	81.87	64.66	80.05	33.61	0.00	0.00	20.47	48.23
MRIQ (single precision)	32k,3k	0.00	0.00	12.47	51.8	10.61	16.70	0.00	0.00	0.00	0.00
MRI-FH (single precision)	32k,3k	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	59.09	59.15
MPEG4 (single precision)	4k,16	45.61	62.04	98.87	87.75	97.69	88.20	0.00	0.00	0.00	0.00
CP (single precision)	32k,3k	0.00	5.36	93.81	30.99	10.58	10.26	0.00	0.00	0.00	0.00
NBODY (single precision)	4k,16	49.36	10.74	62.76	53.59	0.00	0.00	0.00	0.00	0.00	0.00

Table 6.1: Performance percentage loss, resulting from disabling individual optimizations.

manage the register file and improve instruction-level parallelism. For most of the computations, small 2D arrays are accessed from registers and unrolling simplifies access expressions so that the arrays are mapped to registers by the back end compiler.

Copying data to shared memory generally improves performance on both architectures, more significantly on the GTX-280, possibly due to the absence of a data cache as compared to the C2050. TMV and GEMM use shared memory to copy an input matrix into shared memory in coalesced order that is stored in global memory in a different order, which has a huge impact on performance. Table 6.1 shows the impact of texture memory on GEMM. Using texture memory improves performance by up to 30GFlops. While we evaluated use of texture memory on all the benchmarks, we saw little improvement and sometimes degradation on all benchmarks other than GEMM. We make the following observations from our experiments. The gains from texture memory are likely to be more significant on data structures not allocated to shared memory (e.g., register data). We found that texture calls were prohibitively expensive within the core computation, but could be beneficial during the data staging portion of the kernel.

The performance gain in GEMM comes from increased bandwidth to global memory by using the dedicated texture fetch hardware. Texture memory benefits a few other programs, but these same programs improve further by using constant memory. Constant memory on the other hand improves performance in kernels where we are able to put one or more of the appropriately-sized inputs in the constant memory (GEMV,TMV and MRI-FH) that is accessed in the same order across all the threads. Using constant memory eliminates the overhead of shared memory copy and potentially frees up shared memory for shared data in some kernels.

An interesting observation in the table is the absence of all major optimizations in the solution of MRI-FH. Given the number of data structures and their respective sizes, the best performance is achieved by using constant memory for smaller-sized data structures and accessing the larger data structures through global memory in coalesced order.

	Computational Partitioning Phase						Autotuning Phase				
	Total Input	Loop Perms. Pruned		Output	Mult.	Total	Potential	Evaluation		Versions	Total
Kernels	Comb.	Data Dep.	Mem Coal.	Comb.	Kernels	Var.	Variants	Opt.	Params	Eval.	% Pruned
GEMM	6	4	1	1	4	5	7936	124	64	188	97.63
GEMV	2	1	0	1	1	2	560	35	16	51	90.80
TMV	2	1	0	1	1	2	560	35	16	51	90.80
MRIQ	2	1	0	1	1	2	176	11	16	27	84.65
2D-Conv	24	22	1	1	4	5	9984	156	64	220	97.79
MRI-FH	2	1	0	1	1	2	5104	319	16	335	93.43
MPEG4	24	22	1	1	4	5	9984	156	64	220	97.79
CP	6	4	1	1	4	5	2816	44	64	108	96.16
Nbody	2	1	0	1	1	2	304	19	16	35	88.48

Table 6.2: Search space pruning results for computation partitioning phase.

6.6 Size of Optimization Search Space

The section presents the possible sizes of search space and actual number of implementations evaluated for the computations studied in this thesis. Table 6.5 shows the impact of compiler heuristics on pruning the search space of implementations. Dependences and memory coalescing heuristics limit the search space to no more than five different computation partitionings. If we consider all combinations of computation partitioning and data placement decisions, we may end up with thousands of possible scripts. By holding the tile sizes fixed during Phase III of the algorithm, it tries a maximum of 335 transformation recipes. When tuning for optimization parameter values, the total combinations of different values of optimization parameters limits the number of points to search to 64. A total of less than a few hundred strategies are tried to find the final implementation. Almost 97% of the data placement and optimization parameter search space never gets evaluated.

Most of the kernels have majority of their potential variants pruned by the autotuning mechanism, with matrix-matrix multiplication, 2D convolution, MPEG4 and coulombic potential benchmarks benefiting the most with over 95% of the search space pruned. In case of matrix-vector multiplication and transposed matrix-vector multiplication, autotuning system, will save a major effort by pruning the search space by 90.8%. The least benefited from the scheme is MRIQ kernel, with 84% of the search space pruned. However, the total number of possible variants is also significantly lower than some of the other benchmarks evaluated. So the percentage of pruned variants must be seen in the context of total number of possible variants. The larger the potential search space, the higher the benefits of the two-part autotuning system.

6.7 Summary

In this chapter, the results obtained from experimental evaluation of the automatically generated code by our system are presented. We show results for nine computations from BLAS, scientific, imaging and multimedia kernel domains. We show significant improvement on most of the performance numbers achieved by manually-tuned libraries and state-of-the-art compiler generated code. Our results for two different architectures highlight one of the major contributions of this thesis, i.e. performance portability exhibited by the automatically generated code by the system.

Chapter 7

Related Work

This chapter examines the previous and contemporary work related to our research. The following sections will discuss manual optimization, various compiler optimization systems and autotuning frameworks developed to automatically generate GPU code.

7.1 Manually Tuned Computations

While the literature richly describes high-level compiler optimization strategies for targeting architectural features, we have learned through working with application and library developers that users targeting high performance have the perception that commercial compilers fall short in achieving performance levels comparable to what they can derive manually. As a notable contemporary example, GPU programmers aggressively tune their code, due to wide variations in performance of GPU codes resulting from subtle differences. As compared to programming conventional architectures, tuning GPU codes is more difficult and time consuming if done manually. Manual code optimizations for CUDA code have been performed in [BG09, DMV$^+$08, Wol08, VD08a, BCI$^+$08, NTD10, KH10] showing phenomenal performance.

7.2 Transformation Recipe Mechanism

A number of interfaces to code transformations are similar to the transformation recipes in this thesis, including pragma-oriented transformation specifications such as LoopTool [QK06], X language [DBR$^+$05], and Orio [HNS09]. A related tool POET uses an XML-based description of code transformation behavior to produce portable code transformation implementations [YSY$^+$07]. However, our transformation recipes are separate from code, which benefits in keeping the code architecture independent and providing reuse of these recipes for similarly-structured code. A major strength of our system in terms of applying transformations is composition of multiple transformations. We are using the CHiLL framework to compose transformations and generate code, which relies on a polyhedral transformation framework to manipulate mathematical representations of iteration spaces and loop bounds. While other polyhedral frameworks (e.g. [KP93, GVB$^+$06]) provide composition of multiple transformations, the transformation recipes in our system specify high-level transformations that operate on a complete loop nest; transformation algorithms translate from the recipes to the iteration space manipulations for all statements enclosed in the loop nest [Che07, CCH08].

7.3 Annotation-Based Compiler Frameworks

There has been some recent work on compilers generating and optimizing CUDA codes to reduce the complexities of programming GPUs. Lee et al. [LME09] generate CUDA code from sequential C code with extended OpenMP annotations. *hi*CUDA [HA09] is a pragma-based language that translates sequential C code to CUDA code. Both of these annotation-based approaches suffer from limitations in what can be expressed, and therefore impede the compiler from having the flexibility to derive the best implementation. For example, *hi*CUDA code must be explicitly tiled by the programmer, and the pragma must describe the exact footprint for the data to be copied into shared memory.

A few compilers take CUDA programs as input, and produce optimized CUDA as output. CUDA-lite [ULBmWH08] relies on annotations in CUDA code to instruct the compiler how to improve the coalescing of global memory accesses and the bandwidth to global memory. Yang et al. [YXKZ10] can optimize a naive CUDA kernel with thread remapping and data copy optimizations. Their thread remapping approach merges neighboring threads or thread blocks, which can be viewed as a bottom-up version of tiling. Their compiler achieves impressive performance though it still requires the input CUDA

Compilers	R-Stream	PLUTO	Cui et al.	CUDA-CHiLL
PARALLELIZATION STRATEGY				
Blocks	fixed strategy	multiple variants	appears to be fixed	multiple variants
Threads	fixed strategy	multiple variants	appears to be fixed	multiple variants
MEMORY HIERARCHY LEVELS SUPPORTED				
Registers	post-processing	post-processing	$\surd$	$\surd$
Shared Memory	$\surd$	$\surd$	$\surd$	$\surd$
Texture Memory	$\times$	$\times$	$\times$	$\surd$
Constant Memory	$\times$	$\surd$	$\times$	$\surd$
AUTOTUNING SUPPORT				
Optimization Parameters	$\times$	$\surd$	$\surd$	$\surd$
Computation Partitioning Variants	$\times$	$\surd$	$\surd$	$\surd$
Data Layout/Placement Variants	$\times$	limited	$\surd$	$\surd$
Generates transformation recipe	$\times$	$\times$	$\surd$	$\surd$
Pruning Strategy/Measurements	$\times$	$\times$	$\times$	$\surd$

Table 7.1: A comparison of GPU automatic parallelization systems.

kernel to be properly parallelized since it does not support loop permutation. Moreover, manual parallelization may still prove to be cumbersome when dealing with complicated boundary conditions arising from data set sizes that are not even multiples of thread and block sizes.

7.4 Automatic Parallelization for GPUs

A number of automatic parallelization compiler systems have been developed which are based on a polyhedral framework. Compilers based on a polyhedral framework have been proposed to transform sequential codes into optimized CUDA codes, including our work. We summarize each of the compilers in Table 7.1 and describe the code generation algorithm it uses.

The first of these, R-Stream [LVM+10], applies a fixed sequence of optimizations, in keeping with traditional compilers, that are determined through a machine model. It does not perform autotuning, nor does it utilize constant or texture memory.Some optimizations such as unrolling and privatization have to be applied after the code is lowered to the intermediate representation, where our system integrates within the polyhedral framework. These limitations may partially explain why its reported performance results fall far short of those presented in this thesis.

Various aspects of Pluto to support C to CUDA transformation, are described in [BBK$^+$08a, BBK$^+$08b, BRS10]. These papers present individual algorithms such as finding parallel loops for two levels of parallelism with regard to global memory coalescing, finding data footprint size for shared memory copy and finding padding factors to avoid shared memory conflicts. In contrast, we provide a comprehensive optimization strategy generation algorithm to factor in all these factors and their interactions, as well as extensive results to show the impact of autotuning. In [BBK$^+$08b], the autotuning portion and some of the code generation steps are performed outside the polyhedral compiler, which could limit the robustness and effectiveness of the solution. This paper performs limited autotuning: its parallelization algorithm may generate multiple computation partitions arising from global memory coalescing, data placement in shared memory appears to be varied, and a few parameter values tested, but there are few details presented. Further, it is not shown how close their performance is compared to state-of-the-art manual optimization.

The closest work to ours is a script-controlled compilation framework by [CWX$^+$11], designed specifically for generating BLAS kernels. The system takes the script, a predefined optimization scheme, and a data layout plan for matrices, as input to represent a complete optimization strategy. Complementary additional features of the compiler include transposed copy to global memory and padding of triangular matrices. The results reported in their paper are comparable to CUBLAS 3.2 (single precision only). As compared to our results, they use the identical optimization strategy for both the GTX-285 and C2050 for single precision matrix-matrix multiply, while our experience and the experience of the MAGMA library developers [NTD10] is that a very different strategy is needed for C2050 (different computation partitioning, use of texture memory and additional optimizations). The computation partitioning is not described but appears to be fixed. Data layout and mapping variants and optimization strategies appear to try all valid possibilities among a (potentially large) enumerated set. The algorithm to generate optimization strategies, the number of variants generated and strategies to prune variants are not described in detail in the paper. It is not clear how their approach can be expanded beyond BLAS kernels to more general application codes. Further, the compiler does not make use of constant or texture memory. Our compiler, to the best of our knowledge, is the only one that can utilize the full memory hierarchy including the texture memory. With multiple code variant generation and direct support for parameter tuning, we have achieved very high performance, close or in some cases faster than manual tuning. We are also the first to report results on double precision floating point BLAS kernels.

7.5 Autotuning Systems

Some of the older and contemporary systems that use autotuning include self-tuning library generators such as ATLAS [WPD01], PHiPAC [BACD97], OSKI [VDY05] and SPIRAL [PMJ$^+$05]. There are also compiler based autotuners that automatically generate and search a set of alternative implementations of a computation [WP05, GVB$^+$06, BBK$^+$08b]. Application-level autotuners use empirical search across a set of parameter values suggested by the application programmer [CH04].

All of these systems will search multiple code variants to find the best solution for the target architecture. Our autotuning mechanism is distinguished from other autotuning systems because we select code variants and fine-tune optimization parameters in independent phases, greatly reducing the size of search space of potential solutions to be evaluated and still providing comparable or better solutions than manually tuned or other compiler generated code.

7.6 Summary

This chapter presented the previous and contemporary research related to the work presented in this thesis. We examined the research in the discussion about the automatic parallelization for GPUs and autotuning systems.

Chapter 8

Conclusion and Future Work

This chapter summarizes the work presented in this thesis and discusses future work to express possible logical steps that may be taken to extend the system for a wide variety of computations.

8.1 Summary of Thesis

This thesis described a compiler system organized around the concepts of transformation recipes and autotuning which automatically maps a sequential loop nest computation to an equivalent high-performance CUDA implementation for a Nvidia GPUs. The major goals of this system are to provide a set of tools to programmers to generate correct and high performance CUDA code from a sequential computation and save the time and effort that is typically needed to achieve it.

The major contributions of this thesis are the transformation strategy generator (TSG) and the two-phase autonuning system, introduced in Chapter 3 and explained in detail in Chapter 5. The opportunities for parallelization and data placement are explained by the TSG. The TSG, being the main focus of the thesis has been explained in Chapter 5 with key TSG algorithms.

The TSG includes the mechanism to generate multiple strategies with multiple combinations of optimization decisions and optimization parameters, representing different code variants. A number of heuristics, both hardware and software related, are used in the decision making in the TSG. Multiple heuristics act as constraints for many decisions related to both parallelization and data placement. The heuristics and constraints are discussed in detail in the Chapter 4.

The other main contribution, the autotuning mechanism, selects the best performing code variant from the set produced by the TSG. It operates in two phases, which are discussed in Chapter 5, with the description of key algorithms. The independent autotuning of the compiler optimizations and optimization parameters provides a mechanism that navigates the best possible points in the search space, while ignoring the majority uninteresting parts of the search space, resulting in pruning the search space by almost 90% on average for the set of benchmarks studied in for this thesis.

Other than the major focus on the above parts of the system, this thesis covers the interfaces developed or modified with the existing but independent parts of the CUDA-CHiLL system, to make the full parallelizing compiler framework work as a unit. Discussion of this important integration work to make an end-to-end system is given in Chapter 3. It includes the modified interface CUDA-CHiLL uses to map high-level script functions to its the actual implementations. Also discussed is an algorithm for appropriate tile size selection based on occupancy of available resources on a GPU. The algorithm is discussed in Chapter 5 with results presented.

Additionally this thesis makes a key contribution in terms of providing performance portable code by focusing on Nvidia GPUs from two different generations and architectures. The system was tested for a diverse set of benchmarks, on GTX-280 GPU and the C2050 GPU from Nvidia. The results are presented in Chapter 6. This thesis reports performance results for nine computation kernels, from linear algebra, scientific computation and imaging kernels domains. Matrix-matrix multiplication, matrix-vector multiplication and transposed matrix-vector multiplication are important linear algebra routines used in many scientific applications. It is shown that our system generates high performance solutions for these routines which often exceed the performance achieved by the manually tuned CUDA code library from Nvidia i.e. CUBLAS. We have also compared the performance of our system with a contemporary state-of-the-art compiler system called PLUTO, various aspects of which are described in [BBK+08a, BBK+08b, BRS10], on benchmark suite provided by PLUTO. One of the PLUTO benchmarks, MRIQ, was taken from PARBOIL suite [IMP07], and both perform comparable to each other, so our comparison holds for both the versions. Another imaging benchmark evaluated is 2D convolution and the performance is compared to a contemporary compiler system described in [YXKZ10]. The results explained in Chapter 6 show that the code from our systems performs considerably better against PLUTO implementations in its benchmark suite.

Overall this thesis makes a case for autotuning compiler technology as a productivity enhancement for developing high-performance CUDA code for loop nest computations, and porting code from one platform to the next generation. Coupled with the broad set of optimizations, we are able to achieve performance comparable to manually tuned code. What is even more interesting is that in many cases, the compiler-generated code outperforms manually-tuned code and sometimes finds a new optimization strategy. With high-level abstractions for CUDA code generation and an autotuning compiler, the port to the C2050 was relatively straightforward in spite of the architectural differences.

8.2 Future Work

The system developed for this thesis focuses on parallelizing sequential code restricted to an affine domain, where loop bounds and subscript expressions are linear functions of the loop index variables. The underlying software CHiLL will generate correct code when dependences are satisfied. So a future addition to the system is to enable it to handle non-affine codes. Extensions to both the underlying software and system developed for this thesis are required to be able to handle the scenarios stated above. Also, the computations that we have studied in this thesis consist of single loop nests. A system capable of handling multiple loop nests will require additional optimizing and data transfer extensions. We have used a limited set of frequently used optimizations. There are a number of other compiler loop optimizations, available in CHiLL, which may be used as a part of the system like loop splitting, data pre-fetching, skewing and loop fusion. Lastly, some features of CUDA programming model, like concurrent execution of independent kernels, may also be supported by the system for high-performance solutions.

8.3 Summary

This chapter summarized the thesis and revisited the major contributions, including references to the chapters with detailed discussion about them. It summarized the two major parts of the system, the TSG and autotuning mechanism along with the performance numbers for various benchmarks evaluated.

Furthermore, the chapter also discussed some of the logical extensions that can enable the current system to be used for a wide variety of computations using a larger set of compiler transformations and features of the programming model.

Bibliography

[AK02] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, 2002.

[BACD97] Jeff Bilmes, Krste Asanović, Chee-Whye Chin, and James Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, June 1997.

[Ban90] Utpal Banerjee. Unimodular transformations of double loops. In *Proceedings of the 3rd International Workshop on Languages and Compilers for Parallel Computing*, August 1990.

[Ban93] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, 1993.

[Bas04] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, October 2004.

[BBK$^+$08a] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2008.

[BBK$^+$08b] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for GPGPUs. In *Proceedings of the 22nd annual international conference on Supercomputing*, 2008.

[BCI$^+$08] S. Barrachina, M. Castillo, F.D. Igual, R. Mayo, and E.S. Quintana-Orti. Evaluation and tuning of the level 3 cublas for graphics processors. In *Proceedings of the 1st International Parallel and Distributed Processing Symposium*, April 2008.

[BG09] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, 2009.

[BRS10] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Proceedings of the International Conference on Compiler Construction*, March 2010.

[BUH$^+$08] Bondhugula, Uday, Hartono, Albert, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[CCH05] C. Chen, J. Chame, and M.W. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *Proceedings of the International Symposium on Code Generation and Optimization*, March 2005.

[CCH08] Chun Chen, Jacqueline Chame, and Mary Hall. CHiLL: A framework for composing high-level loop transformations. Technical Report 08-897, University of Southern California, June 2008.

[CH04] I-Hsin Chung and Jeffrey K. Hollingsworth. Using information from prior runs to improve automated tuning systems. *SC Conference*, 2004.

[Che07] Chun Chen. *Model-Guided Empirical Optimization for Memory Hierarchy.* PhD thesis, University of Southern California, May 2007.

[CWX$^+$11] Huimin Cui, Lei Wang, Jingling Xue, Yang Yang, and Xiaobing Feng. Automatic library generation for BLAS3 on GPUs. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, May 2011.

[DBR$^+$05] Sebastien Donadio, James Brodman, Thomas Roeder, Kamen Yotov, Denis Barthou, Albert Cohen, María Jesús Garzarán, David Padua, and Keshav Pingali. A language for the compact representation of multiple program versions. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*, October 2005.

[DMV$^+$08] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine A. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC*, page 4. IEEE/ACM, 2008.

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

[HA09] Tianyi David Han and Tarek S. Abdelrahman. *hi*CUDA: a high-level directive-based language for GPU programming. In *Proceedings of the Second Workshop on General-Purpose Computation on Graphics Processing Units*, March 2009.

[HCS$^+$09] Mary Hall, Jacqueline Chame, Jaewook Shin, Chun Chen, Gabe Rudy, and Malik Murtaza Khan. Loop transformation recipes for code generation and auto-tuning. In *LCPC*, October, 2009.

[HNS09] Albert Hartono, Boyana Norris, and P. Sadayappan. Annotation-based empirical performance tuning using Orio. In *Proceedings of the 23rd International Parallel and Distributed Processing Symposium*, May 2009.

[IdFF96] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua an extensible extension language. *Softw. Pract. Exper.*, 26:635–652, June 1996.

[IMP07] IMPACT, 2007. The parboil benchmark suite, 2007. [Online]. Available: http://www.crhc.uiuc.edu/IMPACT/parboil.php.

[KH10] D.B. Kirk and W.W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers, 2010.

[KM92] K. Kennedy and K.S. McKinley. Optimizing for parallelism and data locality. In *ACM International Conference on Supercomputing*, July 1992.

[KP93] Wayne Kelly and William Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, Department of Computer Science, University of Maryland, 1993.

[LME09] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2009.

[LVM$^+$10] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Baskaran, David Wohlford, Cédic Bastoul, and Richard Lethin. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *Proceedings of the Third Workshop on General-Purpose Computation on Graphics Processing Units*, September 2010.

[MCT96] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[NTD10] R. Nath, S. Tomov, and J. Dongarra. An improved magma gemm for fermi gpus. Technical Report UT-CS-10-655 (also LAPACK working note 227), University of Tennessee Computer Science Technical Report, July 2010.

[Nvi08a] Nvidia, 2008. CUDA development Zone. http://developer.nvidia.com/category/zone/cuda-zone.

[Nvi08b] Nvidia. Cuda cublas library, March 2008. http://developer.download.nvidia.com/compute/cuda/2_0/docs/CUBLAS_Library_2.0.pdf.

[PMJ+05] Markus Püschel, José M. F. Moura, Jeremy R. Johnson, David Padua, Manuela M. Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Platform Adaptation*, 93(2):232–275, February 2005.

[QK06] Apan Qasem and Ken Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *Proceedings of the 2006 ACM International Conference on Supercomputing*, June 2006.

[Rud10] Gabe Rudy. CUDA-CHiLL: A programming language interface for GPGPU optimizations and code generation. Master's thesis, University of Utah, May 2010.

[TCC+09] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffery K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Proceedings of the 24th International Parallel and Distributed Processing Symposium*, April 2009.

[TKM+02] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrata nd B. Greenwald, H. Hoffmann, P. Johnson, J. Lee, W. Lee, A. Ma a nd A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, March 2002.

[ULBmWH08] Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen mei W. Hwu. Cuda-lite: Reducing GPU programming complexity. In *LCPC*, pages 1–15, 2008.

[VD08a] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of Supercomputing '08*, November 2008.

[VD08b] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Press, 2008.

[VDY05] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1), 2005.

[WL91] Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[Wol89] M. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, Supercomputing '89, 1989.

[Wol08] Michael Wolfe. Compilers and more: Optimizing gpu kernels. `http://www.hpcwire.com/features/Compilers_and_More_Optimizing_GPU_Kernels.html`, October 2008.

[WP05] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software—Practice and Experience*, 35(2):101–121, February 2005.

[WPD01] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, January 2001.

[YSY$^+$07] Qing Yi, Keith Seymour, Haihang You, Richard Vuduc, and Dan Quinlan. POET: parameterized optimizations for empirical tuning. In *Proceedings of the 21st International Parallel and Distributed Processing Symposium*, March 2007.

[YXKZ10] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A GPGPU compiler for memory optimization and parallelism management. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2010.