CUDA-CHiLL: A PROGRAMMING LANGUAGE INTERFACE FOR GPGPU OPTIMIZATIONS AND CODE GENERATION

by

Gabe Rudy

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

School of Computing

The University of Utah

August 2010

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of **Gabe [redacted]**

has been approved by the following supervisory committee members:

[redacted] Hall [redacted]	, Chair	**5-28-2010** [redacted]
Matthew [redacted]	, Member	**6-16-2010** [redacted]
Matthew Flatt	, Member	**5-28-2010** [redacted]

and by [redacted] , Chair of

the Department of **School of [redacted]**

and by Charles A. Wight, Dean of The Graduate School.

ABSTRACT

The advent of the era of cheap and pervasive many-core and multicore parallel systems has highlighted the disparity of the performance achieved between novice and expert developers targeting parallel architectures. This disparity is most notifiable with software for running general purpose computations on grachics processing units (GPGPU programs). Current methods for implementing GPGPU programs require an expert level understanding of the memory hierarchy and execution model of the hardware to reach peak performance. Even for experts, rewriting a program to exploit these hardware features can be tedious and error prone. Compilers and their ability to make code transformations can assist in the implementation of GPGPU programs, handling many of the target specific details.

This thesis presents CUDA-CHiLL, 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 nonexpert application and library developers.

CUDA-CHiLL is built on the polyhedral program transformation and code generation framework CHiLL, which is capable of robust composition of transformations while preserving the correctness of the program at each step. Through its use of powerful abstractions and a scripting interface, CUDA-CHiLL allows for a developer to focus on optimization strategies and ignore the error prone details and low level constructs of GPGPU programming. The high level framework can be used inside an orthogonal auto-tuning system that can quickly evaluate the space of possible implementations. Although specific to CUDA at the moment, many of the abstractions would hold for any GPGPU framework, particularly OpenCL.

The contributions of this thesis include a programming language approach to providing transformation abstraction and composition, a unifying framework for general and GPU specific transformations, and demonstration of the framework on standard benchmarks that show it capable of matching or outperforming hand-tuned GPU kernels.

To my loving wife

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to specially thank my adviser, Dr. Mary Hall, for her sage guidance, tireless editing and inspirational leadership of myself and her entire research team. This work would be a small pittance of what it became without her dedication to the research and encouraging nature. As a fellow researcher, and my number one user (really only user), I would like to thank Malik Murtaza Khan for running weeks and weeks of benchmarks on the system and collecting much of the results used in this thesis. Naturally, I would like to thank Chun Chen, who laid the groundwork for this thesis with his previous research and ongoing involvement as a research associate. I am fortunate to have Dr. Matt Might and Dr. Matthew Flatt as committee members and I appreciate their involvement and support.

Finally, as special thanks to my loving and supportive wife, who tirelessly read and listened about this work, helped me make schedules and more importantly, helped me keep them.

CHAPTER 1

INTRODUCTION

In the history of microprocessors, the Central Processing Unit (CPU) processor has been the focus of the industry for its powerful ability to run general purpose sequential programs. While hugely successful in meeting the majority of computing needs, there has also been a legacy of programmable accelerators to improve performance for specific domains of applications. In the embedded world, Digital Signal Processors (DSPs) are used to encode and decode audio. In High Performance Computing (HPC), there have been many examples of coprocessors going back to the 1970s designed for floating point calculations or other specific tasks. In consumer computing, discrete video processors have long been included to meet specialized needs of rendering images at the demanding rate of stutter-free video.

Under the pressures of the consumer gaming and professional workstation market, Graphical Processing Units (GPUs) have evolved to deliver ever-increasing amounts of computational performance. Reacting from the market demand to provide more direct means of accessing this potential performance, hardware manufacturers starting providing developer SDKs to treat the GPU as a programmable stream processor, instead of a specialized device only accessible through graphics oriented fixed-function APIs. This opened to door for GPUs to be used for massively parallel computations on nongraphics data. Scientific computing has had a long history of using coprocessors and programmable accelerators to serve its seemingly unbounded need for computational performance. In fact, modern high end super computers often include a hybrid of traditional processors and stream processors in the form of GPUs [56].

1.1 Programmable Accelerators for Scientific Computing

When hardware accelerators of various forms are designed, there is always a trade-off between how general versus function-specific to make the programming model. DSPs, for example, are specialized to handle streams of audio or video data but usually are designed to be programmable to support different and changing encoding standards.

This trade-off can also be seen in the field of Vector Computers such as the Cray 1 in Figure 1.1. The potential speed of the machine was really only unlocked by the use of special vector instructions for doing floating point operations. This required the creation of nonportable programs, but the potential speedup of six to ten times often outweighed the cost.

Other vector computer manufactures added language extensions to high level languages like FORTRAN to make developing for the platform more attractive. The Control Data Cyber 205, for example, made its vector instructions accessible by FORTRAN with notations like `a(1;n) + b(2;n)`, which would add `n` elements of vector `a` starting at position 1 with vector `b` starting at 2.

Eventually, vectorizing compilers that could compile loops to platform specific vector instructions matured to the point that they were good enough for general programming purposes. However, highly tuned libraries were often still written in assembly language to take advantage of the nuanced behavior of the vector instructions. Soon, commercially successful compilers provided feedback and diagnostics to the programmer when a loop was not able to be vectorized. This feedback directed the programmer with specific information as to what was blocking the vecotorization process. With these tools, programmers were then able to write in a portable, high-level language and be certain to have the benefits of vectorized code across hardware vendors.

Figure 1.1. The Cray 1 at the Computer History Museum in Mountain View, CA.

When Intel added the SSE instructions extensions to the x86 family in 1999, there was a similar situation with the vector-based SIMD instructions. At first, manual work was required from programmers to gain access to the potential performance. Eventually, compilers used the existing body of knowledge on vectorizing compilers to generate SSE instructions, when possible, automatically. Again, for programmers seeking this performance, it was key to have some feedback from the compiler on when vectorization was not possible and for what reason.

With GPUs today, we find a lot of similarities to vector computers and SIMD instructions in their requirements on programmers. Platform-specific program changes are required to take advantage of the hardware. Also similarly, although the set of programs that can be adapted with these changes is significant, it will never be universal. In a historical progression, you might expect the next step would be a compiler focused solution that auto-parallizes loops to GPU optimized programs. For various reasons discussed below, the programming of GPUs does not lend itself to this type of automatic and opaque solution.

1.2 Using GPUs for General Purpose Computation

1.2.1 The Early Years

The use of computer graphics hardware for general-purpose computation has been an area of active research for many years. The Ikonas system in 1978 [17], a programmable raster display system for cockpit visualization, allowed the microprogrammable coprocessor to not just handle image processing but flight simulation, ray tracing and solid modeling. The Pixel-Planes group designed a heterogeneous multiprocessor for graphic rendering and “nonscreen” oriented calculations in 1989 [18]. With its independent scheduling of rendering units and message-passing interface between host and device, this general purpose architecture may have been the inspiration for the latest generation of GPUs designed for heterogeneous uses.

Graphics applications themselves have a history of using graphics hardware for procedural texturing and shading [42, 51]. In the early 90s, OpenGL was standardized across platforms and hardware as a way of using GPU resources. Thus, OpenGL was used by researchers as an intermediate language in their abstractions that provided general procedural programming [43, 46]. Clearly, there were still a lot of limitations and unnecessary complexity in using a purely graphics-centric API to gain access to the underlying SIMD capabilities of the hardware. This need was acknowledged by both the API developers and the hardware manufacturers, leading to different but ultimately converging solutions.

1.2.2 GPGPU Frameworks

The two primary hardware vendors in the discrete GPU market, NVIDIA and ATI (now part of AMD), felt the pressure to provide more programmable access to their processors. Clearly, a market was emerging for general purpose computations on non-graphics data on the GPUs. Leading the way with a General Purpose GPU solution (GPGPU), NVIDIA introduced CUDA (Compute Unified Device Architecture) in 2007. AMD also released a similar, but less popular SDK for targeting their GPUs called ATI Stream Technology, based on their own Brook+ compiler and runtime.

With the advent of CUDA, there was an increased use of GPGPU in various research efforts, but consumer providers were reluctant to develop libraries and code based on the proprietary SDKs of a single vendor. In response, a couple of frameworks have emerged to provide a vendor-neutral solution.

Originally developed by Apple and later submitted to the industry consortium that handled the OpenGL specification, Open Computing Language (OpenCL) is a framework for writing programs that can execute on a heterogeneous platform. This framework supports running programs on both GPUs and CPUs, and potentially other accelerator hardware. AMD soon decided to adopt OpenCL as their primary GPGPU framework in favor of their previous Brook+ based system. By 2009, NVIDA had also added support for running OpenCL programs in its updated GPU drivers.

Although OpenCL is gaining more interest for its potential cross platform support, it currently does not run well outside the Mac operating system. Also, by trying to target heterogeneous parallel processing models, OpenCL programs give up some GPU specific optimizations. Currently, an OpenCL program generally performs poorly when compared to a hand-tuned CUDA equivalent.

Microsoft has also developed its own high-level general purpose computing API called DirectCompute, part of DirectX 11 APIs released in 2009. DirectCompute provides a comfortable interface for current Direct3D programmers and abstracts away the vendor-specific hardware. However, its requirement on the DirectX 11 capable GPU has kept it from wide adoption in an industry that tries to target widely adopted technologies.

1.3 Compiler Assisted Targeting of the GPU

It is not hard to see why the GPU is an attractive target for general purpose computing. Figure 1.2 shows that the number of execution units on modern GPUs has grown substantially in the past few years. But unlike earlier targets of vectorizing compilers, GPUs offer a number of challenges to fully exploit their resources.

First, GPUs have two levels of parallelism. At the first level, there is MIMD parallelism across multiple independent streaming multiprocessors (SM). Within each SM, there are multiple thread units that provide SIMD parallelism. Any program that needs to synchronize data between running threads must take into account on which level of parallelism it is operating. While there are built-in methods for synchronizing across SIMD threads, custom and often slow methods are required to synchronize between SMs on the GPU.

Secondly, GPUs have a deep memory hierarchy. Although *global* memory is most commonly used, there is a relatively high latency cost for fetches and stores to global memory. Certain parts of global memory have special performance characteristics such as *constant* and *texture memory*. Outside of global memory, each SM has a certain amount of shared memory locally available. Also, the register file for each SM is large enough to be

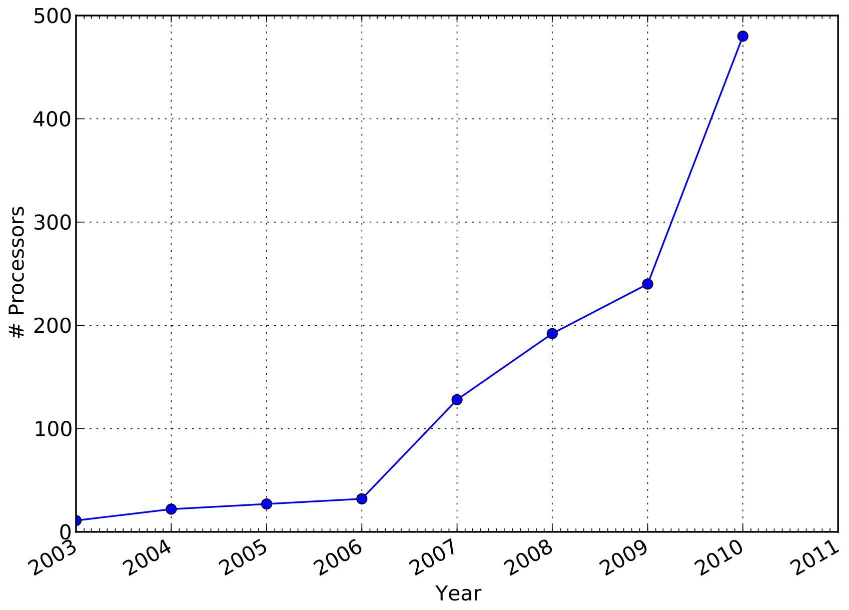


Figure 1.2. # Execution Units (Texture/Vertex Pipelines < 2007) in NVIDA GPUs

considered a memory target for scaler variables in computations.

To target the complex nature of the GPU architecture, programs often have to go through profound transformations. A decomposition of the computational space may be required to match the levels of parallelism on the GPU. Then, to hide the latency of global memory fetches, some data may need to be copied to shared memory or registers. The optimal optimization strategy may not be the obvious one, so various different versions of a GPGPU program may need to be tested and evaluated.

Although it is clear that this process may not be fully automated by a compiler, it is also potentially labor intensive and error prone if unassisted. This thesis proposes that a compiler-assisted solution may be the best for iteratively exploring optimization strategies for the GPU.

1.4 Research Contributions of this Thesis

The contributions of this thesis follow:

- A programming language approach to describing code transformations which provides library and application developers high level abstractions for code generation and compiler transformation capabilities.
- Adding novel transformations and code generations to a unified framework alongside standard compiler transformations in an integrated and composable manner. This framework is capable of generating code targeting GPUs from sequential loop nests by composing these data and computation space partitioning transformations to achieve kernels that match or surpass the performance of hand-tuned equivalents.
- Demonstration of this approach on standard BLAS library routines and some common benchmark kernels yielding results comparable to hand-tuned versions in some cases, outperforming hand-tuned in other cases.

1.5 Organization of this Thesis

This thesis is organized into six chapters. The first two chapters provide background and details of the problem space of optimizing GPGPU programs for today's modern GPU architectures. In Chapter 3, we introduce a programming language approach to directing compiler assistance for this problem space. Chapter 4 provides the details of the system in its underlying technologies and novel components. Then, in Chapter 5, we present benchmarks of the techniques discussed. Finally, we present related work and conclusions in Chapter 6 and 7.

CHAPTER 2

THE GPU AS AN OPTIMIZATION TARGET

As we have seen in the previous chapter, the GPU has progressed to a powerful and programmable parallel architecture, with its multiple streaming multiprocessors with potentially hundreds of cores. Currently, the CUDA framework is the most widely used method of accessing this attractive architecture for general purpose computations. Our goal is to automate many of the difficult programming tasks in generating high-performing, equivalent CUDA code from a sequential computation. The approach presented in this thesis is motivated by the following observations:

- There is a standard and well-defined protocol for GPU kernel computations that involves allocating memory for GPU input and output, copying data to and from the GPU, and performing block and thread decomposition. Given appropriate parameters for computation and data decomposition, these tasks can be automated in a compiler.

- Known compiler transformations can be adapted and applied both in the decomposition and mapping process, and in subsequently optimizing the kernel code to manage the memory hierarchy and parallelism tradeoffs.

- Since there is significant performance variation on GPUs for very subtle differences in code, we would like our system to explore a space of different implementations, and different values of parameters associated with the mapping.

- A programming tool should support both automated compiler optimization as well as programmer-guided optimization, and not get in the programmer's way in achieving high performance.

In the remainder of this chapter, we will describe the salient features of the architecture, the ways in which CUDA accesses these features, and elements of a transformation process

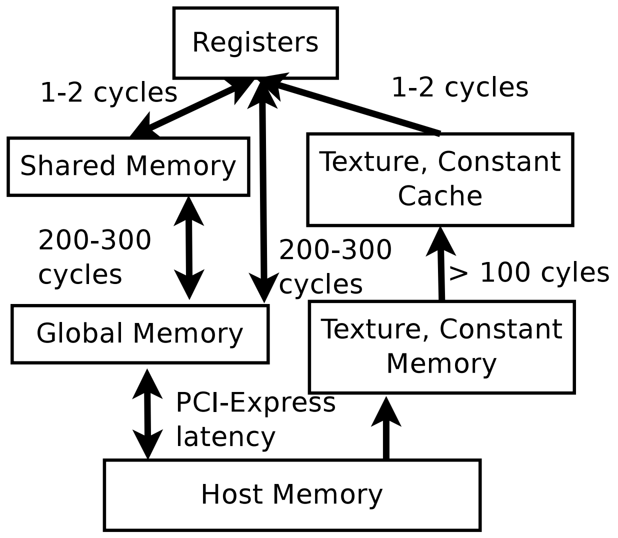


Figure 2.1. The GPU memory hierarchy and latencies between each level.

from a sequential loop nest to a optimized CUDA program.

2.1 GPU Architecture Features

The target GPU architecture for this thesis is an NVIDIA GeForce GTX 280, which is representative of the current generation of GPU offerings from NVIDIA. The GTX 280 is organized into 30 streaming multiprocessors (SM), each of which has an 8-core SIMD unit; the device, therefore, has 240 cores, each clocked at nearly 1.3GHz. Using all cores, a single chip can perform an impressive 933 GFLOPS for single-precision computations, and 78 GFLOPS for double precision. Synchronization between threads in an SM is supported by a barrier; synchronization between threads mapped to different SMs typically relies on atomic operations.

The GTX 280 also has a heterogeneous, and mostly software-controlled, memory hierarchy, consisting of 16K registers and a 16 KByte shared memory *per SM*, and a 1GByte global memory shared among SMs. As seen in Figure 2.1, read-only constant and texture data in global memory are cached for low-latency average access time – but the bulk of global memory accesses are typically not cached, with latencies on the order of hundreds of cycles. To improve bandwidth to global memory, 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 [58].

2.2 Properties of High-Performance CUDA Code

CUDA, introduced in the previous chapter, includes a computing engine, language and compiler tools for running general purpose applications on NVIDIA GPUs [1]. The programming model involves some extensions to the C programming language. These extensions delineate a kernel function to be executed on a GPU, attach memory attributes to variables and provide special syntax to invoke kernel calls. A PathScale Open64 compiler is used to convert the CUDA C program to the NVIDIA GPU native instructions.

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 30 SMs. Each thread block defines a multidimensional space with up to three dimensions and a maximum of 512 threads. A *kernel* code is executed for each point in the grid, providing a two-level parallelism hierarchy represented by this five-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, 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 large size of the register file (16K registers per SM) permits code to exploit significant reuse within a thread. 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 programmer/compiler can order data accesses across neighboring threads (a half-warp) so that global memory accesses are coalesced whenever possible, and shared memory accesses avoid bank conflicts.

While the large body of prior work on locality optimization for conventional architectures sheds some light on compiler optimizations for this and similar GPUs, new challenges arise due to the significant differences between the two levels of parallelism (blocks and threads), and how they synchronize and share data. As mentioned earlier, a low-cost barrier implemented with a CUDA function call permits synchronization between threads.

[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.

Synchronization across blocks is not supported directly in hardware, and is therefore costly and usually avoided.

2.3 Translating Sequential Loop Nests to CUDA

As most computations that are candidates for being run on the GPU start out as nested loops, we look at the process of translating these computations to efficient CUDA code. As we have observed earlier, there is a body of knowledge for compiler transformations that should provide the basis for compiler assistance in this translation process.

As a contemporary example of this translation process, in [70], Wolfe writes about tuning a simple single-precision matrix multiplication kernel on an NVIDIA GeForce GTX 280. Wolfe presents several versions of the matmul code obtained by using code transformations such as loop permutation, loop strip mine and loop unroll, and caching data in local memory. Performance ranges from 1.7 to 208 GFLOPs depending on the number of threads per block, the loop(s) unrolled and unroll sizes, and the amount of data cached in local memory. Summarizing the article, Wolfe writes "Matmul is just one simple example here, three loops, three matrices, lots of parallelism, and yet I put in several days of work to get this seven line loop optimized for GPU."

All seven versions of matmul in [70] can be derived with a combination of loop permutation, strip mine and unroll, and data copy optimization. In the following chapter, we discuss how these standard transformations can be expressed as an optimization strategy in the form of a transformation recipe. Next, we explore in more detail the two fundamental types of transformations made when translating a loop nest to an optimized CUDA program.

2.3.1 Computation Decomposition

Programs written for CUDA must be explicitly aware of the two levels of parallelism of the GPU represented by the two grid dimensions and three block dimensions. It is often the case that the original nested loop computation does not embody the iteration space partitioning to match the GPU index space dimensions. Subdividing the iteration space of a loop into blocks or *tiles* with a fixed maximum size has been widely used when constructing parallel computations [68, 31]. 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 deconstructing 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 constructed control loop then has N/TX steps (when TX divides N evenly). If tiling is used properly, the end result should be a loop nest where some loop levels match the iteration space of the GPU block and thread dimensions for a CUDA kernel. The constraints of loops representing threads and blocks are that they have a stride of 1 and can have their bounds coerced to the fixed iteration space of a GPU grid or block dimension.

In Figure 2.2, we show an example of using tile transformations to do an example deconstruction of the iteration space of a matrix vector multiplication source code shown in Figure 2.2 (a). In Figure 2.2 (b), two tile transformations are done on the first statement of the source code in (a), referenced as statement 0. Both transformations use the tile size of 16 and place the control loop at their default position of being right above the original loop. The tile on line 3 uses the *counted* method, which makes the control loop of stride 1, while the tile on line 5 uses the *strided* method, which results in a control loop with a stride of the tile size. The results given N=1024 and TX=16 can be seen in Figure 2.2 (c), where the outermost *ii* loop is the *counted* control loop and the *jj* loop at level 3 demonstrates a *strided* control loop. The *counted* tile method is used when the objective is to build loops to match the iteration space of GPU thread dimensions.

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

(a) Matrix vector source code (BLAS)

```
1  TX=16
2  l1 = find_cur_level(0,"i")
3  tile(0,l1,TX,l1,"i","ii",counted)
4  l2 = find_cur_level(0,"j")
5  tile(0,l2,TX,l2,"j","jj",strided)
```

(b) Tile transformations as commands

```
for(ii = 0; ii < 64; ii++)
  for(i = 16 * ii; i < 16 * ii + 16; i++)
    for(jj = 0; jj < 1009; jj += 16)
      for(j = jj; j < jj + 16; j++)
        a[i] = a[i] + c[j][i] * b[j];
```

(c) Result of tile command from (b) on input from (a) where N=1024

Figure 2.2. Example of tile transformations to deconstruct iteration space

2.3.2 Targeting the GPU Memory Hierarchy

Other than computation space partitioning, to get optimal performance on a GPU architecture, transformations need to be made to target the deep memory hierarchy. With the high latency associated with a global memory fetch or store discussed earlier, *locality optimizations* should be used to copy data into lower-latency portions of the memory hierarchy.

Similar to how a *tile* operation helps with the computation space decomposition, a *datacopy* operation helps with targeting specific portions of the memory hierarchy. A *datacopy* transformation will introduce a new, smaller dimensional data structure from the array access pattern of statements in a given loop nest. The new data structure can be placed in shared memory and benefit from reuse and low-latency accesses from concurrently running threads.

Various data copy strategies can be used to utilize the shared memory structure accessible to all threads running concurrently on a streaming multiprocessor (SM). Although not an explicitly controlled memory structure, the large register file can be exploited by copying data into fixed sized thread-local arrays and scalars. A variant of *datacopy* variant, which we refer to as *datacopy_privatized*, targets registers by only copying local data touched within a loop nest with respect to a subset of parallelized loops inside the loop nest.

In both cases of targeting shared memory or the register file, it may be necessary to employ a variety of different optimization strategies and empirically test to determine their relative merits. It is therefore extremely useful to have high level constructs that quickly allow for the generation of bug-free implementations of these strategies. In general, the advantage of these operations being done by the compiler are clear. For example, the compiler can ensure correctness by understanding the data dependences between loop statements and handling edge cases of unevenly divisible loop bounds.

In conclusion, although it is somewhat straightforward to convert a loop nest computation to a correct CUDA variant using compiler technology, generating high-performance code requires a sophisticated compiler tool to integrate complex loop transformations for parallelism and flexible data movement to fully utilize all the GPU architecture features that impact performance.

CHAPTER 3

TRANSFORMATION SCRIPTS TO DIRECT OPTIMIZATIONS

The compiler research community has developed a significant body of work in code transformation techniques that improve performance by optimizing for specific architectural features, especially by increasing parallelism or better managing the memory hierarchy [69, 64, 65, 54, 57, 12, 40, 34, 52, 47, 5, 37, 38, 28]. However, the prevailing interface for code optimization on production compilers remains compile-time flags, which limits the utilization of the transformation capabilities of the compiler to optimizations using static analysis. Furthermore, with the complex run-time behavior and hardware interplay of modern processor architectures, it is more effective to evaluate an optimization strategy in a representative execution context. Because of these limitations, application developers are often left with writing optimized code by hand, which is not only difficult and time consuming but results in low-level architecture-specific code that is difficult to port and maintain.

As an alternative to the compile-time flags paradigm, our compilation system exposes its code transformation and code generation capabilities through a *transformation recipe* interface [16, 23]. A transformation recipe expresses an optimization strategy as a sequence of composable transformations. In this form, recipes bring the benefits of separating algorithm writing and architecture specific optimization to application and library developers.

In this chapter, we discuss the use of transformation recipes as a powerful solution to leverage compiler optimization capabilities and achieve very high performance code. We describe the development of transformation recipes in the research community and their limitations in their current form. Finally, we present a more powerful language-based foundation for building layers of abstraction of transformations and provide examples from the CUDA-CHiLL framework.

3.1 Capabilities of Transformation Recipes

3.1.1 Composition of Transformations

In the domain of complex nested loop optimizations, which are quite relevant to scientific and HPC applications, there is a large body of research on loop transformations employed in the optimization process [11, 2, 19]. A key challenge to loop nest optimization is that it is difficult to express and compose a sequence of loop transformations. The ability to express and handle composition is an important feature of an optimization framework because multiple carefully combined program transformations can be necessary to improve performance in many-core architecture with deep memory hierarchies.

For example, after a permutation of the loop order of a loop nest, which is a common transformation to improve memory access patterns, a subsequent transformation must be aware of the difference in the new loop order to perform correctly. Other transformations that manipulate the iteration space may dramatically change the structure of the program. For these reasons, a purely syntactic transformation framework [16, 71, 48] will have difficulties with handling long and complex compositions of transformations in a flexible manner.

More powerful transformation frameworks are based on a polyhedral model of representing a statement's iteration space. Transformations with this model allow for the exploration of alternative iteration spaces while abstracting away many of the implementation artifacts of the syntactic representations. Figure 3.1 (b) shows an example of a transformation recipe using the CHiLL polyhedral framework that performs a long compositions of standard transformations on LU input code in Figure 3.1 (a). The result of applying this recipe is the automatically generated code in Figure 3.1 (c).

Alternative interesting methods of manipulating polyhedral models have been proposed. For example, a purely matrix-operation-based approach was proposed by Cohen et al. [14]. While limiting transformations to those expressible in a purely mathematical manipulation of the polyhedral model, this approach does not resolve the issue of referencing intermediate and newly created semantic constructs.

3.1.2 Auto-Tuning

A well-recognized challenge in code optimizations for modern architectures is making trade-offs between different strategies, or identifying optimal values of optimization parameters such as unroll factors or loop tile sizes. Without sufficient knowledge of the execution environment, which is extremely difficult to model statically, compilers often make suboptimal choices, sometimes even degrading performance. The choice between trade-offs

```
DO K=1,N-1
 DO I=K+1,N
  A(I,K)=A(I,K)/A(K,K)
 DO I=K+1,N
  DO J=K+1,N
   A(I,J)=A(I,J)-
          A(I,K)*A(K,J)
```

(a) Original code

permute([L1,L2,L3])
tile(S1,L3,TJ,L1)
split(S1,L2,[L2≤L1-2])
permute(S2,[L1,L2,L4,L3])
permute(S1,[L1,L3,L4,L2])
split(S1,L2,[L2≥L1-1])
tile(S3,L2,TI_1,L3)
split(S3,L3,[L5≤L2-1])
tile(S3,L5,TK_1,L3)
tile(S3,L5,TJ_1,L4)
datacopy(S3,L4,[D1],1)
datacopy(S3,L5,[D2])
unroll(S3,L5,UI_1)
unroll(S3,L6,UJ_1)
datacopy(S4,L3,[D1],1)
tile(S1,L4,TK_2,L2)
tile(S1,L3,TI_2,L3)
tile(S1,L5,TJ_2,L4)
datacopy(S1,L4,[D1],1)
datacopy(S1,L5,[D2])
unroll(S1,L5,UI_2)
unroll(S1,L6,UJ_2)

(b) Recipe

```
DO T2=2,N,64
 DO T4=2,T2-64,256
  DO T6=1,T4-1,256
   DO T8=T6,MIN(T4-1,T6+255)
    DO T10=T4,MIN(T2-2,T4+255)
     P1(T8-T6+1,T10-T4+1)=A(T10,T8)
   DO T8=T2,MIN(T2+56,N),8
    DO T10=T8,MIN(N,T8+7)
     DO T12=T6,MIN(T6+255,T4-1)
      P2(T12-T6+1,T10-T8+1)=A(T12,T10)
    DO T10=T4,MIN(T2-2,T4+255)
     DO T12=T8,MIN(N,T8+7)
      DO T14=T6,MIN(T6+255,T4-1)
       A(T10,T12)=A(T10,T12)-P1(T14-T6+1,T10-T4+1)
                  *P2(T14-T6+1,T12-T8+1)
  DO T6=T4,MIN(T4+254,T2-3)
   DO T8=T6+1,MIN(T4+255,T2-2)
    P3(T6-T4+1,T8-(T4+1)+1)=A(T8,T6)
  DO T6=T4+1,MIN(T4+255,T2-2)
   DO T8=T2,MIN(N,T2+63)
    DO T10=T4,T6-1
     A(T6,T8)=A(T6,T8)-P3(T10-T4+1,T6-T4+2)*A(T10,T8)
 DO T4=1,T2-65,256
  DO T6=T2-1,N,256
   DO T8=T4,MIN(T4+255,T2-2)
    DO T10=T6,MIN(T6+255,N)
     P4(T8-T4+1,T10-T6+1)=A(T10,T8)
   DO T8=T2,MIN(T2+56,N),8
    DO T10=T8,MIN(N,T8+7)
     DO T12=T4,MIN(T4+255,T2-2)
      P5(T12-T4+1,T10-T8+1)=A(T12,T10)
    DO T10=T6,MIN(T6+255,N)
     DO T12=T8,MIN(N,T8+7)
      DO T14=T4,MIN(T2-2,T4+255)
       A(T10,T12)=A(T10,T12)-P4(T14-T4+1,T10-T6+1)
                  *P5(T14-T4+1,T12-T8+1)
 DO T4=T2-1,MIN(N-1,T2+62)
  DO T8=T4+1,N
   A(T8,T4)=A(T8,T4)/A(T4,T4)
  DO T6=T4+1,MIN(N,T2+63)
   DO T8=T4+1,N
    A(T8,T6)=A(T8,T6)-A(T8,T4)*A(T4,T6)
```

(c) Generated code from (b) with bound parameters

Figure 3.1. Transformation recipe for LU with the CHiLL framework. Recipe syntax simplified for illustration purposes.

can to be made by a developer, a compiler decision algorithm with empirical feedback or a collaboration of the two. A recent body of work on *auto-tuning* uses empirical techniques to execute code segments in representative execution environments to determine the best-performing optimization sequence and parameter values [32, 33, 61, 39, 49, 44, 45, 50, 25].

When parameterized and machine-generated, transformation recipes can form the core foundation of an auto-tuning compiler. Not limited to just simple changes in parameters, auto-tuning compilers can propose multiple code variants representing different optimization strategies that can be compared empirically. A desirable hybrid model involves allowing the developer to guide and interact with the auto-tuning framework, sometimes called *collaborative auto-tuning*.

3.1.3 Remain Architecture and Compiler Agnostic

Transformation recipes allow application and library developers to interact directly with optimization and code generation constructs to transform their code, including expressing parallelism. As proven time and again, hand-coding these changes will result in the undesirable effect of creating a machine-specific program that is difficult to port.

The recipes themselves can be fine-tuned by other developers or modified for different architectures, while the original machine-independent code is maintained with the program. Additionally, using such a system, a developer can now focus on the performance impact of transformations on the target code, instead of the nitty-gritty implementation details of writing correct GPU code.

3.1.4 Interface to Multiple Compilers

The structure of today's compilers makes it difficult to migrate new ideas into practice; retargeting optimizations and decision algorithms for a new compiler infrastructure is often simply infeasible. Every commonly-used compiler infrastructure has strengths and weaknesses, as well as years of development, that are costly to repeat. Converging on one or a small subset of compiler infrastructures is therefore unrealistic.

Composing compiler tools from the collective capabilities of independent systems by ensuring tool interoperability allows for the strengths of each system to be utilized. Transformation recipes can remain compiler and sometimes source language agnostic to be a bridging tool in this interoperable tool chain. A single recipe may even be able to use multiple compiler back-ends to perform the complex set of transformation and code-generation steps.

3.2 A Programming Language Approach to Directing Optimizations

3.2.1 Limitations of a Flat Sequential Transformation Recipe

What have so far been referred to as transformation recipes are flat sequences of transformation commands and parameters. When building a framework that targets a specific architecture, it is necessary to provide layers of abstraction that can automatically handle a variety of input scenarios. Abstraction layers allow the optimization strategy to work at a level of detail more general than the ultimate target architecture.

To build such a rich and effective abstraction layer, the following traits are desirable.

- **Parameters as variables** Mutable variables allow for parameters to be set by the *auto-tuning* framework or by other methods.
- **Queryable program state** Queries capture the side-effects, results of commands and other code state.
- **Control flow** Branches and iteration constructs react to captured or parameterized information.
- **Encapsulation** Group together commands that are context insensitive to previous commands.
- **Readability** Reference semantic constructs in a way that carries over to the final generated program.

With these goals, it becomes immediately apparent that we are thinking of transformation recipes as programs written in a very limited language. Function outputs, logical branching and the ability to build layers of abstraction and reusable, generic components are easily obtained by using a more fully featured language. In doing so, we no longer refer to these objects as *transformation recipes*, but *transformation scripts*.

3.2.2 Scripting Language Foundation for Compiler Scripts

Although there may be a few promising candidates when selecting a language on which to base a compiler framework for expressing code transformations, the Lua [27] programming language should certainly be on the short list.

Lua is a lightweight, embeddable scripting language with extensible semantics and easy integration with a host program. With its small set of general features, it can be extended to fit a variety of different problem types. In fact, Lua has been described as “multiparadigm” with features that allow for functional, imperative and object-oriented programming styles.

Lua syntax is also very familiar. Transformation recipes of existing frameworks should be convertible to a script of Lua functions with very minimal syntax changes. Moving to a standard and easily embeddable language also allows for scripts to be potentially universal to more than just a single compiler framework if the compiler community were to establish a standard API for common transformation commands.

3.3 CUDA-CHiLL: A Compiler Framework for Generating Optimized CUDA Code

In this section, we introduce CUDA-CHiLL, a unified framework for novel transformations and code generation targeting CUDA code. The CUDA-CHiLL framework is capable of transforming sequential loop nests with standardized loop transformations and data movement commands. The commands are issued by a Lua script and can include complex compositions and use of abstraction layers. When the script execution is finished, CUDA-CHiLL outputs the resulting CUDA source program.

3.3.1 Basis on CHiLL

CUDA-CHiLL expands upon the standardized loop transformation commands of CHiLL [2]. A polyhedral transformation framework, CHiLL provides a powerful collection of polyhedral transformations for handling computation partitioning targeting parallelism on multicore architectures and data copy and layout operations that target deep memory hierarchies. CHiLL uses a modified version of the Omega Library [29] for polyhedral manipulation and scanning.

The following is a summary of some of the additional functionality that CUDA-CHiLL provides compared to CHiLL.

- **Lua scripting bindings** CHiLL uses a sequentially executed custom recipe language to provide its transformation and code generation constructs to the script writer.

- **Use of loop variables as semantic handles** CHiLL commands use a statement group and loop level to reference specific loop constructs. As each statement modifies the shape and order of the polyhedral space, each command’s addressing must reflect the current state of the program. By adding loop index variables to CUDA-CHiLL, more context-free groups of transformations and abstraction layers can be written.

- **New introspective functions** To allow for scripts to handle a variety of input code and parameters, a number of functions were added to query the framework's internal representation of the polyhedral model, index variables and CUDA mappings.

- **CUDA-specific functions and code generation** New commands were needed to generate CUDA-specific constructs and allow the code generation to output valid CUDA programs. These include directing the mapping of loops to grid indices and data transfers between the host and GPU. Also, support was added for thread synchronization calls and special variable attributes like shared memory data structures.

3.3.2 Loop Variables as Semantic Handles

The ability to reference newly created semantic constructs is important to support composition of multiple transformations and also to encapsulate groups of transformations in a context-independent manner. CHiLL's current method of addressing constructs in its polyhedral model uses a statements group number and a loop level. A *tile* command, for example, will produce a new loop level. Subsequent transformation commands must take into account this change in the internal representation of the polyhedral model when they make references using statement numbers and loop levels. The result of this addressing method can be hard to follow recipes such as the LU transformation recipe in Figure 3.1 (b). For example, `L4`, `L5` and `L6` refer to loops created by tile and split commands. Although powerful in its ability to compose multiple transformations, it is almost necessary to print out generated code at each line in the recipe in order to see what constructs are being referenced in the next command.

We make the observation that a loop index variable uniquely identifies a nested loop for a given statement. Thus, instead of specifying loop levels, the index variable for that level can be used. But as a loop *level* may reference different semantic constructs with each change to the polyhedral model, a loop *index* is a sticky handle that stays valid as new loop levels are introduced or others removed. As an additional advantage, having to specify a fresh index variable for newly created loops provides both a handle for referencing the loop in later operations and a more readable output from code generation.

For example, Figure 3.2 shows a transformation script applied to matrix vector multiply that introduces two new control loops to be used as CUDA block and thread indices, `ii` and `jj`. In Figure 3.2 (b), these new loops are used as handles for the CUDA mapping operation at line 7 and they can be seen as the loop indices of the first two loop levels of the intermediate code shown in (c).

3.4 Abstraction Layers with Lua

Another feature of the underlying scripting language of CUDA-CHiLL that allows for shorter, more powerful and easily readable transformation scripts is the ability to write usable abstraction layers. Although every script can use the functions provided by CUDA-CHiLL directly to express the optimization strategies, building a domain-specific abstraction layer allows for shorter scripts and the reuse of common patterns.

Such abstraction layers themselves may collect the best strategies for targeting a specific architecture, which may change over time as better optimization techniques arise. By being specific to an architecture, simplifying assumptions can be made that allow the script writer a proper level of detail for constructing their optimization strategies.

In CUDA-CHiLL, the high-level functions described in the next section are written in Lua and built using the functions provided by CUDA-CHiLL. At the top of a transformation script, a Lua command `dofile(hlcuda.lua)` is used to import these functions into the current script namespace.

3.4.1 High Level Description of Tiling

With the CUDA-CHiLL `hlcuda.lua` abstraction layer, the function `tile_by_index` provides a powerful and quick way to direct the computation space decomposition of the original nested loop. As shown in usage in Figure 3.3 (c), `tile_by_index` can express a group of tile operations and loop permutations through a high level Lua function. The original code is shown in Figure 3.3 (a) and the generated code in (d). Table 3.1 provides descriptions of each parameter of the `tile_by_index` command.

An algorithm discussed in Chapter 4 is employed to determine the tile operations and loop order permutations required to achieve an end result that matches the list of index variables given as the final parameter to `tile_by_index`. As shown when comparing the original CHiLL tile commands in Figure 3.3 (b) to the CUDA-CHiLL equivalent in Figure 3.3 (c), `tile_by_index` is often performing more than one tile transformation per function call.

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

(a) The matrix-vector sequential source code.

```
1  init("seqMV.suif", 0, 0)
2  dofile(hlcuda.lua)
3  N=1024
4  TI=16
5  tile_by_index({"i","j"}, {TI,TI},
     {l1_control="ii", l2_control="jj"},
     {"ii", "jj", "i", "j"})
6  print_code()
7  cudaize("gpuMV", {a=N, b=N, c=N*N},
     {block={"ii"}, thread={"jj"}})
```

(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++)
        a[i] = a[i] + c[j][i] * b[j];
```

(c) The results of output at line 6 from (b).

```
void seqMV(float **c, float *a, float *b)
{
float *devO1Ptr, *devI1Ptr, *devI2Ptr;
cudaMalloc(&devO1Ptr, 1024 * 4);
cudaMemcpy(devO1Ptr, a, 1024 * 4,
    cudaMemcpyHostToDevice);
cudaMalloc(&devI1Ptr, 1048576 * 4);
cudaMemcpy(devI1Ptr, c, 1048576 * 4,
    cudaMemcpyHostToDevice);
cudaMalloc(&devI2Ptr, 1024 * 4);
cudaMemcpy(devI2Ptr, b, 1024 * 4,
    cudaMemcpyHostToDevice);
dim3 dimGrid(64, 1);
dim3 dimBlock(64, 1);
gpuMV<<<dimGrid,dimBlock>>>(devO1Ptr,
    devI1Ptr, devI2Ptr);
cudaMemcpy(a, devO1Ptr, 1024 * 4,
    cudaMemcpyDeviceToHost);
cudaFree(devO1Ptr);
cudaFree(devI1Ptr);
cudaFree(devI2Ptr);
}
```

(d) The resulting CUDA scaffolding in the original host-executed function.

```
__global__ void gpuMV(float *a, float **c,
    float *b)
{
  int bx = blockIdx.x; int tx = threadIdx.x;
  for (i = 16 * bx; i < 16 * bx + 16; i++)
    for (j = 16 * tx; j < 16 * tx + 16; j++)
      a[i] = a[i] + c[j][i] * b[j];
}
```

(e) The resulting CUDA kernel

Figure 3.2. A simplified example of tiled and CUDAized matrix-vector multiply

Table 3.1. Description of prominent commands in CUDA-CHiLL scripts.

Command	Example Parameter	Description
`tile_by_index`	`{"i","j"}`	The index variables of the loops that will be tiled
	`{TI,TJ}`	The respective tile sizes for each index variable
	`{l1_control="ii", l2_control="jj"}`	A mapping that specifies control loop variable names and optionally renames tile loop index variables.
	`{"ii", "jj", "i", "j"}`	Final order of nested loops with update loop index names
`cudaize`	`"gpuMV"`	The name of the kernel function
	`{a=N, b=N, c=N*N}`	The data sizes of the arrays if not statically determinable
	`{block={"ii"}, thread={"jj"}}`	Block and thread indices for mapping. The bounds for these loops are used to define the grid dimensions.
`copy_to_registers`	`"kk"`	The loop level, given as an index variable, that is the target of register structure
	`"c"`	The name of the array variable to be copied
`copy_to_shared`	`"tx"`	The loop level, given as an index variable, that is the target of the copied data
	`"b"`	The name of the array variable to be copied
	`-16`	Ensure the dimensions of the temporary array are coprime with 16
`unroll_to_level`	1	Unrolls all statements up to one level from innermost loops outwards. This construct will stop unrolling if it encounters a CUDA thread mapped index.

3.4.2 High Level Description of Data Copy

The `hlcuda.lua` abstraction layer also provides functions for *locality optimizations* that target shared memory and registers. These functions, `copy_to_shared` and `copy_to_registers`, are based on CUDA-CHiLL's *datacopy* and *datacopy_privatized* operations, discussed more in Chapter 4, but utilize other features of CUDA-CHiLL to meet their objectives.

For example, `copy_to_shared` must ensure that the resulting extracted loops match one of the defined CUDA dimensions and that stores to the shared memory are properly synchronized. For targeting registers, the CUDA C compiler will predictably place local array accesses into registers providing the index expressions are simply determinable. Thus, *copy_to_registers* unrolls loops introduced by *datacopy_privatized* to remove variables from the array index expressions.

```
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)

```
original()
tile(0,1,TI,1,counted)
tile(0,3,TJ,2,counted)
tile(0,5,TK,3,strided)
tile(0,4,TK,4,counted)
tile(0,5,1,5,counted)
tile(0,5,1,4,counted)
```

(b) CHiLL tile commands

```
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"})
```

(c) CUDA-CHiLL tile_by_index equivalent

```
for (ii = 0; ii < 16; ii++)
  for (jj = 0; jj < 64; jj++)
    for (kk = 0; kk < 1009; kk += 16)
      for (t = 0; t < 16; t++)
        for (tt = 0; tt < 4; tt++)
          for (j = 16 * jj; j < 16 * jj + 16; j++)
            for (k = kk; k < kk + 16; k++)
              c[j][64 * ii + 16 * tt + t] = c[j][64 * ii + 16 * tt + t] +
                 a[k][64 * ii + 16 * tt + t] * b[j][k];
```

(d) Generated code after tile commands

Figure 3.3. Initial tiling for matrix multiply for GPU parallelism (CUBLAS 2)

As seen in Figure 3.4, the resulting use of these high level functions to target the deep memory hierarchy of a GPU is much shorter and expressive than the CHiLL equivalent. The commands used in Figure 3.4 (b) have their parameters explained in Table 3.1. Note that the CHiLL recipe shown in Figure 3.4 (a) does not handle CUDA-specific constructs such as designating shared memory and inserting synchronization barriers but is otherwise comparable in the resulting transformations.

Figure 3.5 depicts how the computation of a small N=8 matrix-vector multiply (MV) problem shown in Figure 2.2 (a) can be optimized with these high level data copy commands. A *copy_to_shared* command is used to copy sections of the *b* vector into shared memory using multiple threads. A *copy_to_registers* is used to copy individual values picked by the thread and block index (*tid* and *bid*, respectively, in the figure) from the *a* result vector to a *tmp* variable where it is used in the computation and then copied back to the result vector. Although minimal in its problem size and tile size for example purposes, this is the same

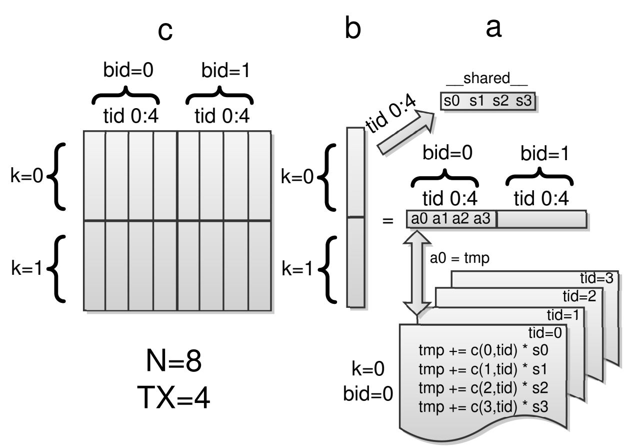
```
--register copy
datacopy_privatized(0, 3, "c", 4, 5, false, -1, 1, 1)
--shared memory copy
datacopy(0, 4, "b", false, 0, 1, -16)
--thread parallelism for shared memory copy
tile(3,4,(TJ*TJ)/TI,4,counted)
tile(3,6,1,4,counted)
--unroll register copy loops
unroll(1,5,0)
unroll(2,5,0)
--fully unroll shared copy loop
unroll(3,6,0)
--finally unroll main computation
unroll(0,8,0)
unroll(0,9,0)
--unroll additional cleanup loops
unroll(8,6,0)
unroll(4,5,0)
unroll(6,5,0)
```

(a) CHiLL commands

```
copy_to_registers("kk", "c")
copy_to_shared("tx", "b", -16)
unroll_to_depth(2)
```

(b) CUDA-CHiLL equivalent where "kk" and "tx" are index variables

Figure 3.4. Data copy and unroll for matrix multiply for GPU memory hierarchy (CUBLAS 2)



```
__shared__ float s[4];
float tmp = a[4*bid+tid];
for (k = 0; k < 2; k++)
{
  s[tid] = b[4*k+tid];
  __syncthreads();
  tmp += c[4*k+0][4*bid+tid] * s[0];
  tmp += c[4*k+1][4*bid+tid] * s[1];
  tmp += c[4*k+2][4*bid+tid] * s[2];
  tmp += c[4*k+3][4*bid+tid] * s[3];
  __syncthreads();
}
a[4 * bid + tid] = tmp;
```

Figure 3.5. Given an easy to depict, although trivially small problem size of N=8 for matrix-vector multiplication, the decomposition using a tile size of TX=4 is visually depicted with the global memory matrix, vectors and shared memory structures. The corresponding kernel code is also given.

strategy used in the MV script discussed in Chapter 5 for running benchmarks.

3.4.3 High Level Description of Unroll

Part of a CUDA optimization strategy is sufficient granularity that the performance benefits of running on the GPU outweigh the overhead of data transfer and thread execution. This often means leaving some loop levels inside the kernel. Unrolling these loops in the GPU kernel provides the same performance benefits that unroll provides on conventional processors, except because of some GPU attributes discussed below, these benefits may be even more notable on the GPU.

Because hardware vendors try to provide as many execution units as possible in a GPU processor, the complexity of each execution unit is fairly simple. Without much silicon devoted to branch prediction, and with the lock-step manner in which threads are executed inside a SM, unrolling sometimes improves performance purely by removal of loop iteration overhead. Also, the compiler may be able to reuse data in registers and improve instruction scheduling when compiling statements from unrolled fixed-size loops.

The `unroll_to_level` command unrolls the innermost loops of compute statements. It may unroll up to a specified number of levels, but never unrolls a loop mapped to a thread or block. Also inherent in this CUDA-specific abstraction is the careful treatment of extra clean-up statements often created by the unroll transformation. These statements may not be created for every combination of data set size and tile dimensions. But, when generating library code for all potential problem sizes, various dimensions will result in different sets of clean up statements. In this case, `unroll_to_level` makes certain to have the iteration spaces of the clean-up statements also mapped to CUDA thread dimensions for optimal performance.

Because `unroll_to_level` can query to detect side effects, this approach dramatically simplified generating optimized code for every potential matrix size of the `matmul` case study discussed in Chapter 5. Earlier research iterations required individually handling the sequence of `unroll` operations required for the different variations of clean-up loops generated by the various problem sizes.

3.5 Summary

With the high level abstraction interface and its specialization in building optimized GPU code, CUDA-CHiLL exemplifies the utility of using scripting languages as the interface to compiler transformations and code generation. Applicable transformations such as tiling and loop unrolling can be made available at a level of detail suited to a developer composing an optimization strategy. Next, we discuss the details of CUDA-CHiLL behind the abstraction interface.

CHAPTER 4

CUDA-CHILL TRANSFORMATIONS AND CODE GENERATION

In this chapter, we present how the organization of our compiler system falls into two phases. Phase I builds an internal representation of the source program and performs transformations and queries on that representation. Phase II of the compiler generates optimized code for the CUDA platform for various problem sizes and optimization techniques.

4.1 The Two Phase Structure

To allow for the most flexible and powerful model of code transformations and target-specific optimizations, CUDA-CHiLL's compiler framework is partitioned into two phases of operation, as shown in Figure 4.1. Phase I is governed by the running of a Lua transformation script. The start of the script loads and initializes the internal representation of CUDA-CHiLL based on an input source file. Each API call of the script performs a transformation or query on this internal representation.

When the script finishes its execution, the framework enters Phase II. With the newly transformed and updated internal representation, a code generation library is given groups of statements and constraints and attempts to generate as output the cleanest sequential code representation for the statements. This intermediate generated code is then specialized for the CUDA architecture with the details of the CUDA grid space and data movement gathered from running the transformation script. Finally, the output is ready to be translated to compilable CUDA C that should be functionally equivalent to the input source program.

4.1.1 Interpreting Transformation Scripts

Executing transformation scripts in CUDA-CHiLL involves creating, updating and querying an internal representation of the source program. When initialized at the beginning of a script, CUDA-CHiLL is given a source file that is preprocessed into a high-level source

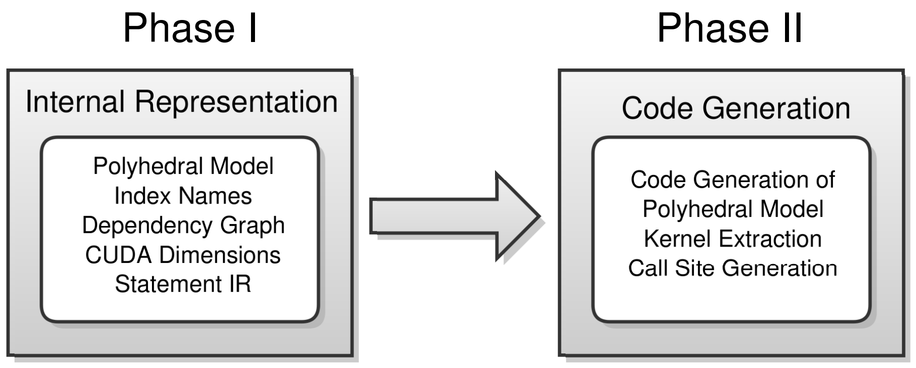


Figure 4.1. CUDA-CHiLL System Diagram

intermediate representation (IR) based on SUIF [62]. Along with the SUIF source file, a specific procedure and loop are given as the target of the transformation script.

At initialization, the input IR for the specified loop is transformed into a set of groups of statements where each group shares the same index space. The internal representation of CUDA-CHiLL then consists of the following items.

- The original IR for the statements themselves, which may be updated and finally used in Phase II to reconstruct program code.
- A polyhedral model of the iteration space and transformations on that iteration space for each group of statements.
- Ordering of statements at each dimension of the polyhedral space.
- A data dependence graph between statements.
- A set of index variables per statement as handles to dimensions of the polyhedral model (and ultimately the loops that could be generated from those dimensions).
- Details to transform the output to parallel GPU code. This includes grid dimensions, details for kernel function extraction and memory management between the host and GPU.

Through API calls made in the Lua script, this internal representation is updated and queried. As we discussed in Chapter 3, polyhedral models for program transformations provide the most flexibility in allowing for the composition of many transformations while ensuring a valid representation of the program. CUDA-CHiLL builds on CHiLL's polyhedral model for representing the iteration space of groups of statements. Transformations such as *tile* and *datacopy* introduced in Chapter 2 mutate the polyhedral model and potentially the other states that compose the internal representation.

Special commands such as *cudaize* described in more detail in this chapter may not perform transformations to the polyhedral model but provide the binding information to be used later in Phase II of the framework.

4.1.2 Code Generation

Once the Lua transformation script finishes execution, Phase II of the framework converts the internal representation into the optimized output source code. Along with the polyhedral model for each statement, other internal representation is used to generate sequential code that uses specified index variables for the generated loop levels. Also, given details on how the computation will be translated to a GPU kernel, the code generation is instructed not to split up iteration spaces that need to be mapped to a single GPU grid dimension.

At this stage, the code is generated as a single nested sequential loop. Although CUDA-CHiLL focuses on targeting CUDA C as the destination language and framework, this last stage of the compiler could be parameterized for other GPU language targets such as OpenCL or even multicore CPU targets such as OpenMP or pthreads. From the sequential nested loop, the provided GPU mapping information is used to extract a kernel and generate a call site that is inserted in place of the original loop in the final output code.

4.2 Phase I

Once the internal representation of the input code is initialized, there is a number of functional interfaces to transformation and query commands in CUDA-CHiLL. Many transformation commands operate on a given statement group (often thought of as just a statement) and a dimension of the polyhedral iteration model for that statement. Since the dimensions of the polyhedral model are always ordered in terms of the nesting level of the equivalent loop representation, a single dimension of the model can be referenced by its nesting depth (level).

More conveniently and for reasons previously discussed, since each nesting level corresponds to an index variable, and commands that introduce new levels must also specify a new index variable, the index variable itself can be used in transformations. Thus, with the pair of a statement number and an index variable, most transformations have a named handle that uniquely identifies a program object.

4.2.1 Querying the Internal Representation

To build powerful abstraction layers such as `hlcuda.lua` introduced in the previous chapter, some functions were introduced to provide useful run-time querying into the internal representation of the input code. These functions could be used at any time to dynamically query the various transformation states caused by different problem sizes or parameters. For example, since `unroll` may or may not require the creation of clean-up statements, `num_statements` could be used before and after a call to `unroll` to determine if new statements were created that need extra handling. Table 4.1 lists the functions and their description.

4.2.2 CUDA Mapping Semantics

To enable a transformation script to target a transformed nested loop to a GPU, a transformation construct must be introduced to specify the mapping from well-formatted loop nests to GPU dimensions. In CUDA-CHiLL, the `cudaize` command provides the necessary information to extract from the sequential loop produced in Phase II a separate *kernel* function that will execute as GPU threads. The compiler also generates scaffolding to invoke this kernel and manage the data movement from the host to the GPU and back for relevant data structures for the computation. Figure 3.2 (d) shows an example of this

Table 4.1. Description of CUDA-CHiLL querying commands

Function	Description
`num_statements`	Retrieves the total number of statements
`cur_indices`	Retrieves the index variable names of each loop level as a list for a given statement
block_indices `thread_indices`	Retrieves a list of the mapped indices for the block or thread grid dimensions as specified by an earlier `cudaize` call.
block_dims `thread_dims`	Retrieves a list of the integral dimensions of the block or thread dimensions as specified by an earlier `cudaize` call.
hard_loop_bounds	Returns the lower and upper bounds if available for a given statement group and loop. -1 returned for when hard bounds are not calculable.
`does_exist`	Returns whether a given variable exists in the procedure or global symbol tables

generated scaffolding and (e) shows its related generated kernel function.

In the Table 3.1, we describe the details of the parameters used in the `cudaize` command. Notice the indices that specify which dimensions of the polyhedral space are placeholders for the CUDA grid dimensions. These indices will be explicitly renamed to the appropriate abbreviated CUDA index name from the set `bx,by,tx,ty,tz`. We later discuss how other CUDA-CHiLL operations utilize this naming convention to intelligently map their own computation space to CUDA grid dimensions. Also, these index variables are used during code generation to transform the generated sequential loop nest to CUDA code.

4.2.3 Additional Block and Thread Index Mapping

Because PHASE II of CUDA-CHiLL uses the specially named loop index variables to direct which loops are to be replaced by CUDA grid indices, new statements created after the *cudaize* call can also have CUDA grid indices. For example, transformations like `datacopy` and `unroll` create new statements. For some of these new statements, like in the case of an `unroll` clean-up statement, index names will be properly inherited. These include CUDA grid index names that will be replaced by dimensions of the CUDA grid space during code generation. For others, especially those created by `datacopy`, statements may need to be tiled to match the appropriate grid dimensions. Finally, some of the index names for the iteration space of these new statements must be explicitly set to have the statements be properly executed in a parallel manner on the GPU.

CUDA-CHiLL provides a `rename_index` command that given a statement, loop level and variable name, will explicitly set the loop index variable for code generation. In the case of the given name being one of the grid dimension abbreviated names, the effect of the renaming is that the specified loop will be mapped to that grid dimension just like the ones specified by `cudaize`.

4.2.4 Other Available Transformations

CUDA-CHiLL expands upon the transformations in CHiLL to provide all the necessary tools for targeting the CUDA framework. Table 4.2 describes some of the common commands used in the CUDA-CHiLL framework by higher level abstractions like the one provided by `hlcuda.lua`. These commands have been updated with CUDA-specific features and the use of index variables as semantic handles.

High level functions use these commands, as well as other CUDA-CHiLL commands such as the querying interfaces, to build their powerful abstractions for the target architectures.

Table 4.2. Description of common CUDA-CHiLL commands

Command	Description
tile	Tile a given loop with a given tile size. Provide an index variable for the newly generated control loop and second index variable to optionally rename the original index variable in the tile loop.
permute	Reorder the loop nest for a given statement according to a given loop order. The order is specified by a list of index variables.
datacopy	For a specified statement and starting loop level, copy array access of a specified array to a smaller dimensional structure. Optionally annotate the new data structure with `__shared__` to specify a copy to shared memory.
datacopy_privatized	Similar to `datacopy` but used to copy data private to a thread and thus doesn't have an option to flag for shared memory.

Figure 4.2 and Figure 4.3 present the details of the algorithms used in the `tile_by_index` and `unroll_to_level` high level functions introduced in the previous chapter.

4.2.5 Adding Synchronization of Shared Memory Structures

Shared memory data structures can be used concurrently by threads running on the same SM. But before a thread examines memory locations in a shared data structure that were written by other threads, a call to `__syncthreads()` is necessary.

The `copy_to_shared` high level function discussed in Chapter 3 creates a copy of data in shared memory to achieve better reuse and lower latency access. The high level function uses the `add_sync` command to insert the synchronization calls during code generation. `add_sync` takes a statement group number and index name to identify the loop level containing writes to shared memory. The `__syncthreads()` command is then inserted after this loop during code generation.

4.3 Phase II

4.3.1 Directing the Polyhedral Code Generation

Building on the foundation of CHiLL's code generation features that in turn utilize the Omega Library [29], CUDA-CHiLL must handle CUDA-specific constructs and constraints when converting the internal representation to a sequential loop nest. For example, CUDA uses special attributes, such as `__global__`, to identify kernel functions and memory classifications of data structures that are not valid ANSI C. Also, left to its own constraints, the code generation libraries sometimes split up the generated loops based on the polyhedral

```
TileByIndexCommands(s,I,S,M,O)
Input: $s$: Statement number; $I$: Indices to tile; $S$: Tile sizes;
       $M$: Map of names for indices; $O$: Final loop nest order
Output: $F$: Set of transformation operations
begin
  $F$ := $\emptyset$
  $C$ := extract control loop name list from $M$
  $I'$ := extract renamed tile loop name map from $M \cup I$
  $order$ := $BuildOrder(O, C, I', 0)$
  $F$ := $F + [\![$permute($s$,$order$)$]\!]$
  for $i$ in $1..|I|$ do
    $level$ := $FindLevel(I_i)$
    $order$ := $BuildOrder(O, C, I', i)$
    $offset$ := offset between $I'_i$ and $C_i$ in $order$
    if $offset < 0$ then
      $F$ := $F + [\![$tile($s$,$level$,$S_i$,$level + offset$,$I'_i$,$C_i$)$]\!]$
    then
      $F$ := $F + [\![$tile($s$,$level$,$S_i$,$level$,$I'_i$,$C_i$)$]\!]$
    end
    $order$ := $BuildOrder(O, C, I', i)$
    $F$ := $F + [\![$permute($s$,$order$)$]\!]$
  end
  return $F$
end
```

```
BuildOrder(O,C,I,n)
Input:
  $O$: Final loop nest order;
  $C$: Control loop list;
  $I$: Index loop mapping;
  $n$: Current index
Output: $B$: Built order
begin
  $B$ := {}
  $C'$ := $C$ after position $n$
  for $o$ in $O$ do
    Skip if $o \in C'$
    if $o \in I$ then
      $o$ := $I(o)$
    end
    $B$ := $B :: o$
  end
  return $B$
end
```

Figure 4.2. Algorithm used by `tile_by_index` where *FindLevel* finds the loop level of an index variable in the current internal representation and *BuildOrder* builds the snapshot of what the order should be between its current state and its final state given n tile operations were already processed.

```
UnrollToLevelCommands(l)
Input: $l$: maximum levels to unroll;
Output: Set of transformation operations
begin
  $F$ := $\emptyset$
  repeat
    for each statement $s$ do
      for $l'$ innermost to outermost levels of $s$, stopping when reaching a thread
      index level or iterated $l$ times do
        $F$ := $F + [\![$unroll($s$,$l'$)$]\!]$
      end
    end
  until no new statements are introduced
  return $F$
end
```

Figure 4.3. Algorithm used by `unroll_to_level` model for multiple statements to produce readable and minimally gated code. CUDA-CHiLL must then specify when a dimension of the polyhedral model must be contiguous for the purposes of later transformations of the output code to parallel GPU code.

4.3.2 Transforming Sequential to Parallel Code

As we previously outlined, after the internal representation is directed through the generation of a sequential loop nest, transformations are performed on this code IR to produce the desired GPU code.

As an example of what this transformation looks like, consider the CUDA code shown in Figure 3.2(d). This code is generated to provide the proper scaffolding to handle data movement and make the GPU kernel call. The array references in the sequential loop nest are analyzed to determine their read and write properties. Appropriate `cudaMalloc` and `cudaMemcpy` calls are then generated for each array to transfer data to and from GPU *global memory*. If size attributes were specified in the `cudaize` call during the running of the transformation script, they will be used in these memory copy operations.

The kernel call requires parameters to define the CUDA grid space used when executing the kernel. In this case, there is only one grid dimension and thread dimension (all other dimensions are set to 1). The call to the generated GPU kernel, `gpuMV`, is made with the `dim3` variables that define the execution grid space as extra parameters. CUDA uses the `<<< >>>` syntax in a C++-template manner prefixing the parameter list at the call site of the kernel function with these dimension variables.

Finally, as an effect of the `cudaize` call in the transformation script, for each statement group, there should be loop levels with specially renamed index variables from the set `bx,by,tx,ty,tz`. During the transformation to CUDA code, loops with these indices are removed and references to these variables are replaced with the CUDA provided index variables from the set `block.x, block.y, thread.x, thread.y, thread.z`. If there was a compound upper bound for a removed loop, it is replaced with a bounds check to ensure correctness. For example, if the upper bound for the `tx` loop was `min(-(58 * bx) + 116, 128)` and the `thread.x` grid dimension was 128, the loop construct would be replaced with the condition `if(tx < -(58 * block.x) + 116)`.

4.4 Summary

With the desired GPU code produced, CUDA-CHiLL has completed Phase II. The output code is ready to be compiled by the CUDA C compiler and will be functionally equivalent to the original sequential code but execute on the GPU. As part of an auto-tuning or library-building strategy, CUDA-CHiLL can rapidly produce many code variants based on different problem sizes or optimization parameters. Its high level abstraction layer and low level constructs discussed in this chapter can be intermixed to provide the ultimate flexibility in construction transformation scripts that produce high performance GPU code.

CHAPTER 5

EVALUATION

In this chapter, we show how CUDA-CHiLL, combined with insights on how to optimize CUDA for the target GPU architecture, can be used to generate high-performance GPU code. As we will show in some examples, this generated code often matches or exceeds the performance of manually tuned libraries. We describe the optimization strategy for the three BLAS kernels matrix-vector multiply, transposed-matrix-vector multiply and matrix-matrix multiply. BLAS kernels are well studied and their deceptively simple nature allows for the full range of transformation choices. They thus provide a good baseline for comparing different optimization strategies, including those vetted by previous research and adopted in high performance libraries.

5.1 Optimization Heuristics

Before analyzing each individual kernel, we first provide some compiler optimization heuristics that are used to design the optimization strategies embodied in the transformation scripts described later in this chapter. These heuristics could also be used in future work to design an automated compiler decision algorithm for generating scripts that represent the potential search space of optimization strategies. In combination with an auto-tuning framework, such a system may find strategies for existing or new code that exceed the performance of the current method of expertly constructed strategies.

As discussed in Chapter 2, there are multiple architecture features at play in any given optimization strategy. We make the observation that these three BLAS kernels have significant parallelism and few data dependences (only on the accumulation to the output), so strategies that maximized memory hierarchy performance are by and large safe from a parallelization perspective. Although it is also necessary to consider computation space partitioning when looking to optimize for the deep memory hierarchy of GPUs. Locality optimizations assume a computational parallelism is already in place to be able to target relevant memory structures.

We therefore considered multiple areas of research targeting parallel architectures and locality optimizations to be able to define the optimization heuristics and the transformation scripts that use them in this chapter. These include manual tuning strategies for GPUs [59, 9, 6], compiler approaches for combining parallelism with locality optimization (*e.g.,* [31]) and memory hierarchy optimization strategies that compose multiple optimizations [13, 40].

The memory hierarchy optimization approach is an adaptation of the algorithm in [13]. However, it is also significantly influenced by the work of Volkov et al. in optimizing matrix-matrix multiply for the NVIDIA GTX, which was incorporated into the CUBLAS 2.0 library [59]. The key insight from the algorithm in [13] 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 levels of memory hierarchy in the target GPU: register, shared memory and global memory. [1] Unlike a conventional cache-based memory hierarchy where different memory hierarchies have an order of magnitude difference in size, the GPU's register file size collectively in a thread block is larger than the size of shared memory. Locality optimizations should take this into account when defining loop ordering.

The following is a list of the optimization heuristics for high performance GPGPU code.

- **H1: 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.

- **H2: 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

[1] We do not use constant memory or texture memory for these codes, following Volkov et al.

different threads into shared memory in a coalesced order, and accessed directly from shared memory.

- **H3: Shared Memory and Bank Conflicts** Data shared across threads, either as a result of the global memory access coalescing optimization above, or through significant inherent reuse, are placed in shared memory. Shared memory accesses need to avoid bank conflicts along the thread index, and two-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.

- **H4: Maximize Reuse in Registers** Registers provide low latency storage local to a thread, and data that are reused within a thread can benefit by being placed in 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.

5.2 BLAS Library Benchmarks

Note that the CUBLAS library obeys the convention of standard Fortran BLAS implementations: "For maximum compatibility with existing Fortran environments, CUBLAS uses column-major storage and 1-based indexing" [41]. Since C uses row-major storage, the two-dimensional matrices are implicitly transposed for the CUBLAS calls. In this section, we use the CUBLAS naming, so, for example, what we refer to as "transposed-matrix-vector multiply" is not actually transposed.

5.2.1 Matrix Vector Multiply

Figure 3.2(a) and Figure 5.1 (a) shows matrix vector multiply source code and its CUDA-CHiLL transformation script, respectively. Following H1, there is a reduction on loop j but no dependence on loop i, so loop i is tiled to provide multiprocessor and thread-level parallelism (lines 2 and 3). Following H2, there is no reuse of array c and with the i loop parallelized, its accesses to global memory are already coalesced, so it remains in global memory. After the computation space decomposition, the transformed code is matched to CUDA block and thread indices in line 3. Now we use only one block dimension (for unsynchronized parallelism) and one thread dimension (for synchronized data sharing). This effects a change in the naming of the specified indices in later parts of the script.

Following H3, array *b*, which exhibits reuse across threads, is copied to shared memory in line 5. Inside the `copy_to_shared` construct, additional tiling of the new parallel copy statement is done to follow the thread partitioning of the main computation. Following H4, line 6 holds the privately computed array *a* for each thread in registers. To reduce loop overhead, line 7 fully unrolls the main computation loop. This unroll may create extra cleanup loop structures that will automatically be configured to match the thread parallisim of the main loop and be unrolled themselves if necessary. It turns out that properly handling these cleanup loops which occur for not evenly divisible tile sizes has a huge performance impact on the problem sizes that require them.

Figure 5.1(b) shows the simplest generated CUDA kernel from this script, when array sizes are divisible by the tile size of 64. Notice the vector notation, such as `P1[0:15]`, that is an abbreviation for multiple source lines of code in the generated output.

5.2.2 Transpose Matrix Vector Multiply

Figure 5.2 shows transpose-matrix-vector multiply source code (a) and associated CUDA-CHiLL script (b). The loop nest is parallelized in the same way as MV, as shown in lines 2 and 3. The CUDA mapping is also the same as MV in line 5. We can also exploit reuse on *b* by copying to shared memory in line 5.

Array *c* in transpose-matrix-vector multiply is not accessed in a coalesced order from global memory. So even though *c* has no reuse within the thread block, we copy to shared

```
1  dofile(hlcuda.lua)
2  tile_by_index({"i","j"}, {TI,TI},
       {l1_control="ii", l2_control="k"},
       {"ii", "k", "i", "j"})
3  normalize_index("i")
4  cudaize("mv_GPU", {a=N, b=N, c=N*N},
       {block={"ii"}, thread={"i"}})
5  copy_to_shared("tx", "b", 1)
6  copy_to_registers("k", "a")
7  unroll_to_depth(1)
```

(a) CUDA-CHiLL script for MV

```
float tmp;
__shared__ float P1[16];
bx = blockIdx.x;
tx = threadIdx.x;
tmp = a[tx+16*bx][0];
for (k = 0; k <= 63; k++) {
  P1[tx] = b[16*k+tx];
  __syncthreads();
  tmp += c[16*k:16*k+15][16*bx+tx]
         *P1[0:15];
  __syncthreads();
}
a[tx+16*bx] = tmp;
```

(b) Generated CUDA kernel code from above script (N=1024,TI=16)

Figure 5.1. MV transformation recipe and generated code.

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

(a) Transpose-matrix-vector multiply source code (BLAS)

```
dofile(hlcuda.lua)
tile_by_index({"i","j"}, {TI,TI}, {l1_control="ii", l2_control="k"},
        {"ii", "k", "i", "j"})
normalize_index("i")
cudaize("tmv_GPU", {a=N, b=N, c=N*N}, {block={"ii"}, thread={"i"}})
copy_to_shared("tx", "b", 1)
copy_to_shared("tx", "c", -16)
copy_to_registers("k", "a")
unroll_to_depth(1)
```

(b) CUDA-CHiLL script for TMV

```
float tmp;
__shared__ float P1[16];
__shared__ float P2[16][17];
bx = blockIdx.x;
tx = threadIdx.x;
tmp = a[tx + 16 * bx];
for (k = 0; k <= 63; k++) {
  P1[tx] = b[16 * k + tx];
  __syncthreads();
  P2[0:15][tx] = c[16 * bx:16 * bx + 15][16 * k + tx];
  __syncthreads();
  tmp += P2[tx][16 * k:16 * k + 15] * P1[0:15];
  __syncthreads();
}
a[tx + 16 * bx] = tmp;
```

(c) Generated CUDA kernel code from above script (N=1024,TI=16)

Figure 5.2. Transformation recipe and generated code for TMV.

memory in coalesced order by different threads so that in the computation, the threads access the data from shared memory. Line 6 performs this shared memory copy. The last parameter to `copy_to_shared`, -16, is used in the datacopy transformation to pad the temporary array size to be coprime with 16, which is the number of banks in shared memory, to avoid shared memory bank conflicts.

Again, we unroll the main computation in line 8. This time, the `unroll_to_depth` abstraction also handles unrolling the inner loop of the shared memory copy code for c. Because of the extra shared memory copy, there may be even more clean-up loops generated for the various problem and tile sizes for this code transformation that will be optimally handled. An example of optimized code generated from the above script is shown in Figure 5.2(c).

5.2.3 Matrix Multiply

Figure 5.3 shows the matrix multiply source code (a) and CUDA-CHiLL transformation script (b). For matrix multiply, array c has dependences carried by the k loop, and it has the most reuse, so it is the best candidate to be kept in fast registers and thus, the k loop is kept as the innermost loop during computation space partitioning (see lines 2, 3 and 4). The main computation loop is partitioned with the tile size $\texttt{TJ} \times \texttt{TI/TJ}$. The CUDA mapping uses two block indices and two thread indices in line 5.

We apply the register copy transformation on line 6 to calculate the correct temporary array footprint with regard to parallelized loops access of c. An unroll is implicitly done on the loop that copies into and out of the this local array so that the compiler will see it as a target for register allocation. Line 7 is for the shared memory copy, similar to the TMV example above, where b is copied in a coalesced order by one set of threads, and accessed in a different order from shared memory. Again, there will be some tiling of the shared memory copy loop to use match the thread parallelism of the computation.

The `unroll_to_depth` abstraction is instructed to unroll up to two loop levels in line 8. The main compute loop with its j and k indices will be unrolled fully, as well as the shared memory copy loops. Finally, the initial unrolls will be creating clean-up loops that themselves will be properly mapped to threads and unrolled.

Figure 5.3(c) shows one resulting CUDA kernel from the above script. The resulting code closely matches the Matrix Multiply code in [59] except for a slightly different parallel mapping scheme for the shared memory copy code.

```
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)

```
dofile(hlcuda.lua)
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"}, strided)
cudaize("mm_GPU", {a=N*N, b=N*N, c=N*N},
    {block={"ii", "jj"}, thread={"t", "tt"}})
copy_to_registers("kk", "c")
copy_to_shared("tx", "b", -16)
unroll_to_depth(2)
```

(b) CUDA-CHiLL script for MM

```
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 (kk = 0; t10 <= 1008; kk += 16) {
  P2[tx][4 * ty:4 * ty + 3] = b[16 * by + 4 * ty:16 * by + 4 * ty + 3][tx + kk];
  __syncthreads();
  P1[0:15] += a[kk + 0][64 * bx +16 * ty + tx] * P2[0][0:15]
  P1[0:15] += a[kk + 1][64 * bx +16 * ty + tx] * P2[1][0:15]
  ...
  P1[0:15] += a[kk + 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 above script (N=1024,TI=64,TJ=TK=16)

Figure 5.3. Transformation recipe and generated code for MM.

5.3 Performance

We applied the recipes from the previous section to the sequential code for the three BLAS kernels: matrix-vector multiplication (MV), transposed-matrix-vector multiplication (TMV) and matrix-matrix multiplication (MM). We measured performance of each kernel against the CUBLAS 2.0 library released by Nvidia for a range of problem sizes with square matrices ranging from 128 to 8192 elements for each kernel.

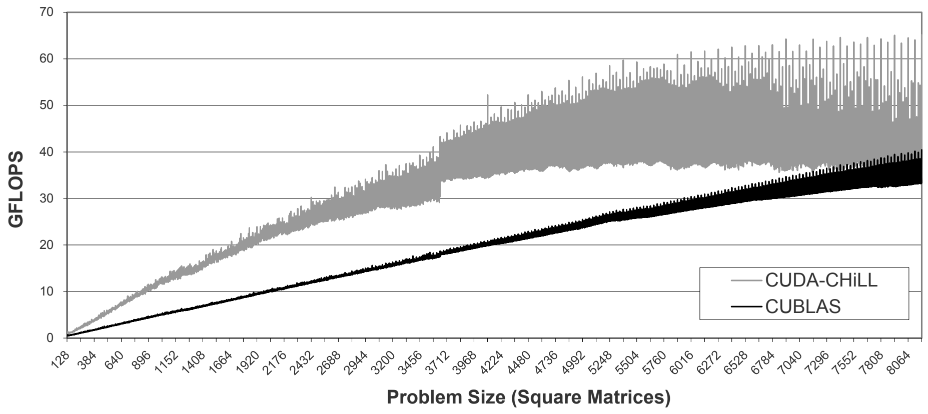
We used auto-tuning in a limited way to identify tile sizes that performed best for a each matrix size with MV and TMV and a small set of tile sizes for the more computationally expensive MM. The size and layout of the tile determines the number of threads active in a block, and the total number of threads. Further, when tiling is used with data copy, tile size dictates amount of shared memory used per block or registers used per thread. With these considerations, we constrained the tile sizes to relatively small numbers, and multiples of 8.

Once tile sizes were chosen, each version of the generated code and the CUBLAS library was run 3 times, and we used the average of those runs recording time using CUDA events. We graph the results for all data points on the three kernels in Figures 5.4 and Figures 5.5.

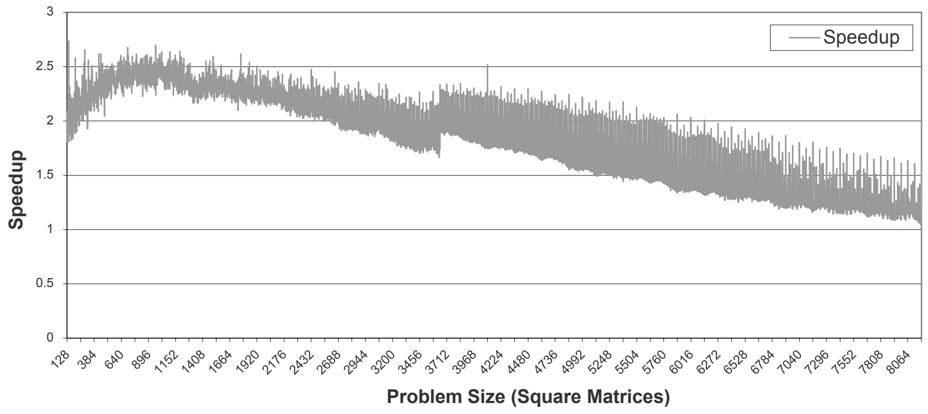
Figures 5.4(a) and (b) compare the performance of the CUDA-CHiLL automatically-generated code with CUBLAS for MV, and the speedup of CUDA-CHiLL over CUBLAS, respectively, within the range of matrix sizes. On average, CUDA-CHiLL provides a 1.8x speedup over the CUBLAS libraries. While more speedup was achieved for smaller problem sizes, with a maximum of 2.7x, the CUDA-CHiLL automatically-generated code never fell below the performance of the CUBLAS libraries. The tile sizes chosen for each matrix size varied from 8 to 120. In previous experiments, a fixed tile size of 16 was used for all problem sizes, but as the problem sizes went over 2k, performance suffered in comparison to the CUBLAS libraries.

Figure 5.4(c) and Figure 5.5(a) present the performance comparison between CUDA-CHiLL and CUBLAS for TMV. The automatically-generated code stays very close to the performance of the library numbers. It excels in the smaller problem sizes, with a maximum of 1.7x speedup and falls to a minimum 0.8x the library performance in a few larger problem sizes. Again, various tile sizes are tested for each problem, with sizes ranging from 8 to 56 chosen based on performance.

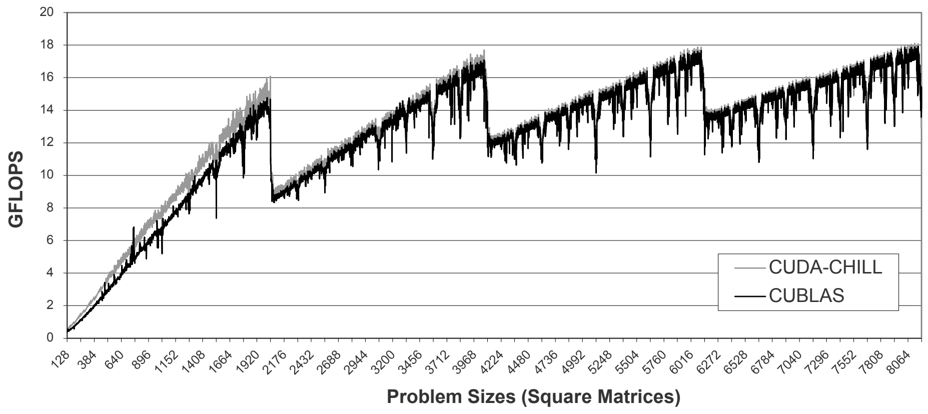
Figures 5.5(b) and (c) compare performance between CUDA-CHiLL and CUBLAS for MM. The performance is consistently better with the CUDA-CHiLL code, with an average 1.5x speedup over the library code and a maximum 2.5x speedup. CUBLAS achieving 372



(a)MV GFLOPS

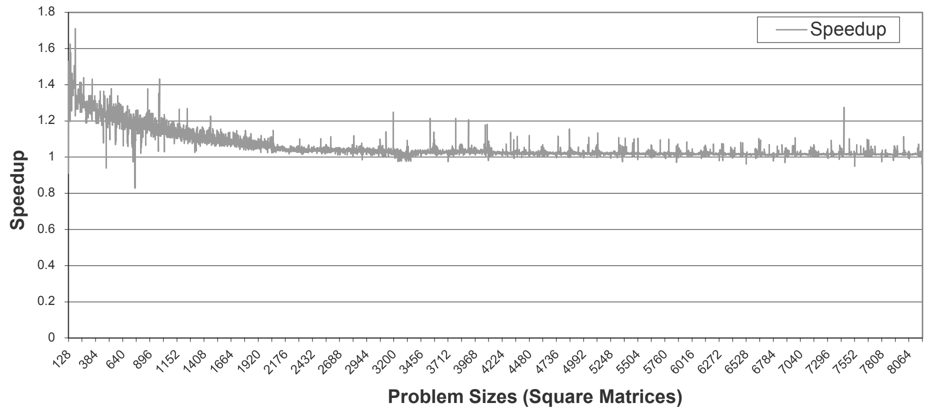


(b)MV Speedup

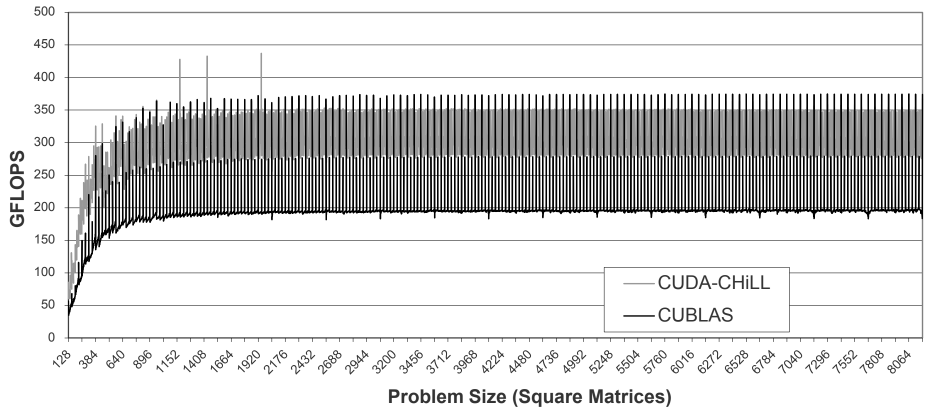


(c)TMV GFLOPS

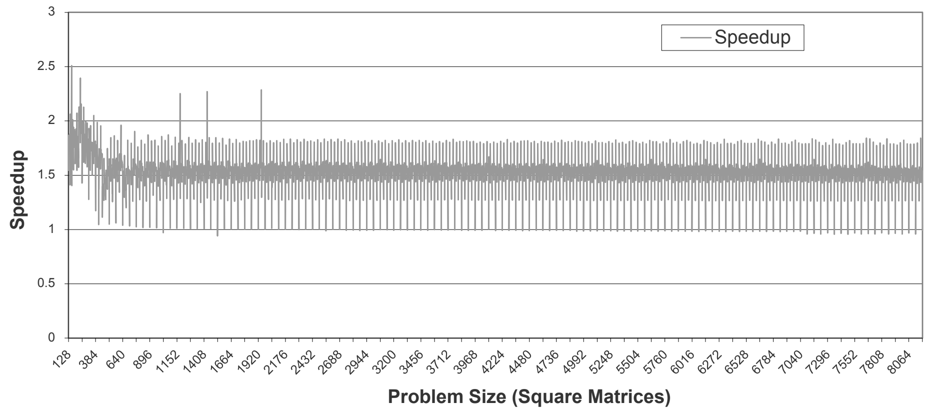
Figure 5.4. Performance comparison of automatically generated code with CUBLAS 2.0 for matrix-vector (a,b), transposed-matrix-vector (c) multiplication kernels.



(a)TMV Speedup



(b)MM GFLOPS



(c)MM Speedup

Figure 5.5. Performance comparison of automatically generated code with CUBLAS 2.0 for transposed-matrix-vector (a) and matrix-matrix (b,c) multiplication kernels.

GFLOPS at its highest, wile CUDA-CHiLL is able to achieve 435 GFLOPS. While it is significant that compiler-generated code can beat the performance of manually-tuned code, an even more significant accomplishment of our compiler is the difference in performance for all other matrix sizes that are not multiples of 16, where the CUBLAS library performance drops by 40-50%. There were, however, some points in the range where the CUDA-CHiLL performed only 0.9x the library code. In our experiments, we used relatively few tile size centered around 64x16. Further performance improvements may be achieved by testing a wider selection of tile sizes and potentially different optimization strategies for specific problem sizes.

From this experiment, when coupled with the heuristics of the previous section, we have learned that various capacity and architectural features point to a small set of strategies for generating highly-tuned GPU code. We plan to generalize this work to increase the level of automation, incorporating the heuristics of the previous section with the constraints on tile sizes we learned from these experiments into a compiler decision algorithm that relies on auto-tuning to fine-tune tile sizes.

CHAPTER 6

RELATED WORK

While the literature richly describes high-level compiler optimization strategies for targeting architectural features, as we have discussed in Chapter 3, the prevailing interface of compile-time flags falls short of achieving performance levels compared to what can be derived 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. Manual code optimizations for CUDA code have been performed in [9, 15, 70, 59, 6], showing phenomenal improvement in performance.

The research in this thesis and the CUDA-CHiLL system are based on the idea of a transformation script that performs source-to-source transformations. It uses a polyhedral model to allow for powerful composition of transformations, and it is specialized to the targeting of GPU multicore and deep memory hierarchy architecture features with CUDA-specific code generation. In this chapter, we progress through the related research in each of these topics upon which CUDA-CHiLL builds.

6.1 Source Transformation Frameworks

A number of works related to describing compiler transformations are similar to the transformation scripts in this thesis. A common model for describing transformations involves inlining the description of the transformation into the original source code itself as pragmas such as in LoopTool [48] or as well-formed comments such as with Orio [25].

The X language [16] allows transformation commands to exist inline with tagging pragmas in the source or in a separate file. It uses pragma and macro substitution to perform predefined or custom pattern-matching-based rewrite rules on source syntax. A related tool, POET uses an XML-based description of code transformation behavior to produce portable code transformation implementations [71].

As discussed in Chapter 3, syntax-focused transformation systems have a hard time expressing long compositions of transformations, especially those that introduce new semantic elements. The separation of the transformation recipe or script from the original source code provides the benefit of keeping the source code architecture independent and increases the potential for the transformation script to be reused with similarly-structured code.

6.2 Polyhedral Loop Transformations

A major strength of CUDA-CHiLL in terms of applying transformations is composition of multiple transformations. CUDA-CHiLL builds on CHiLL and inherits its polyhedral transformation framework to manipulate mathematical representations of iteration spaces and loop bounds.

The PLuTo [11] compiler framework and others [30, 20] also use a polyhedral model for source to source transformations. These low-level frameworks allow for multiple transformations to be performed, but often fall short of providing high level interfaces to the transformations. CUDA-CHiLL incorporates both high-level transformations that operate on complete loop nests and other target relevant commands affecting code generation.

6.3 Parallelization and CUDA Targeting

The basic components of building a transformation framework that targets the GPU architecture are subcomponents that partition computation to achieve higher levels of parallelism and align or copy memory to perform on the deep memory hierarchy. A compile-time transformation scheme [3] has shown efficient global memory access can be achieved by employing program transformations. CUDA-lite [58] focuses on achieving this optimization automatically by using simple annotations to improve the coalescing of global memory accesses and the bandwidth to global memory.

There is a nontrivial amount of complexity just in provisioning of the correct program constructs of a complex algorithm written in CUDA. To help with this problem, there has been research on the CUDA code generation given an already parallel program source. These include a framework to convert a parallel OpenMP source program to CUDA [7] and a pragma-based approach to auto-parallelization of sequential loops [24] .

Although not a transformation framework, PyCUDA [4] provides ways of limiting the amount of CUDA code to just the compute kernel function itself by providing Python libraries for code generation of call-site scaffolding as well as interfaces to the CUDA C compiler.

CHAPTER 7

CONCLUSION

This thesis presents a programming language interface to GPGPU optimizations and code generation. GPUs provide an attractive target for running parallel computations with the potential for order of magnitude speedups over conventional CPU implementations. Because of the complex nature of the GPU architecture, it is often very difficult to construct programs that utilize the deep memory hierarchy of the GPU for maximum performance. Although compilers are capable of many useful transformations of loop nests that can be applied to target the parallel execution units and memory hierarchy of the GPU, current ways of utilizing these transformations are difficult to compose and lack constructs for generating GPGPU code. We present CUDA-CHiLL as a unifying framework to transform complex loop nests to highly optimized NVIDIA CUDA programs using a programming language approach. We show that this approach can match or outperform the performance of hand-tuned, standard math libraries.

Although CUDA-CHiLL currently targets CUDA during its code generation phase, the high level transformations and even GPU memory structure targeting are completely relevant to other GPGPU frameworks, specifically OpenCL. Most computations benefit the most from explicitly-managed data movement between uncached global memory and low latency shared memory. For computations that have data reuse but less determinable access patterns, further research could be done in incorporating other memory structures such as the cached texture memory into CUDA-CHiLL. Due to its automated nature and programmable interface, CUDA-CHiLL could easily be a component in a larger compiler infrastructure. For example, an *auto-tuning* framework that targets GPUs could leverage CUDA-CHiLL for generating code variants that reflect various optimization strategies or sets of parameters that get continuously narrowed through empirical feedback. It may also be desirable to build even higher level abstractions for this purpose. One could imagine OpenMP-like parallelization directives that, in turn, use complex scripts to target loop nests to GPUs.

Because it separates the optimization strategy from the sequential source code on which it operates, CUDA-CHiLL permits expression of the code in a high-level architecture independent way. This advantage could be realized when targeting new architectures with a potentially different balance of hardware features. Also, heterogeneous platforms share the complexity problem with GPUs. High level transformation abstractions can allow for targeting these platforms while leveraging existing optimization strategies such as those described in this thesis.

Finally, another advantage of keeping the transformation script separate from the target source code is the ability to reuse scripts on multiple programs. Already, scripts could be written to be fairly program agnostic, as index names can be queried and used as variables, new index names can be generated without symbol conflicts and scripts can have divergent behavior based on the program it is transforming. Further research could explore ways of making scripts even more general or creating libraries of common optimization strategies as functional snippets.

REFERENCES

[1] `http://developer.nvidia.com/object/cuda.html`.

[2] `http://www.cs.utah.edu/~chunchen/`.

[3] `http://rosecompiler.org/`.

[4] `http://documen.tician.de/pycuda/`.

[5] Ahmed, N., Mateev, N., and Pingali, K. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Proceedings of the 2000 ACM International Conference on Supercomputing* (May 2000).

[6] Barrachina, S., Castillo, M., Igual, F., Mayo, R., and Quintana-Orti, E. Evaluation and tuning of the level 3 cublas for graphics processors. In *Proceedings of the 1st International Parallel and Distributed Processing Symposium* (Apr. 2008).

[7] Baskaran, M. M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., and Sadayappan, P. A compiler framework for optimization of affine loop nests for GPGPUs. In *Proceedings of the 22nd annual international conference on Supercomputing* (New York, NY, USA, 2008), ICS '08, ACM, pp. 225–234.

[8] Baskaran, M. M., Vydyanathan, N., Bondhugula, U. K. R., Ramanujam, J., Rountev, A., and Sadayappan, P. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2009), PPoPP '09, ACM, pp. 219–228.

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

[10] Blume, W., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T., Lee, J., Padua, D., Paek, Y., Pottenger, B., Rauchwerger, L., and Tu, P. Parallel programming with polaris. *Computer 29*, 12 (Dec. 1996).

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

[12] Carr, S., and Kennedy, K. Improving the ratio of memory operations to floating-point operations in loops. *ACM Trans. Program. Lang. Syst. 16*, 6 (Nov. 1994), 1768–1810.

[13] Chen, C., Chame, J., and Hall, M. 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* (Mar. 2005).

[14] Cohen, A., Sigler, M., Girbal, S., Temam, O., Parello, D., and Vasilache, N. Facilitating the search for compositions of program transformations. In *Proceedings of the 19th annual international conference on Supercomputing* (New York, NY, USA, 2005), ICS '05, ACM, pp. 151–160.

[15] Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., and Yelick, K. A. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC* (2008), IEEE/ACM, p. 4.

[16] Donadio, S., Brodman, J., Roeder, T., and Yotov, K. A language for the compact representation of multiple program versions. In *Lecture Notes in Computer Science* (2006), IEEE Press, pp. 136–151.

[17] England, J. N. A system for interactive modeling of physical curved surface objects. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1978), SIGGRAPH '78, ACM, pp. 336–340.

[18] Fuchs, H., Poulton, J., Eyles, J., Greer, T., Goldfeather, J., Ellsworth, D., Molnar, S., Turk, G., Tebbs, B., and Israel, L. Pixel-planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1989), SIGGRAPH '89, ACM, pp. 79–88.

[19] Girbal, S., Vasilache, N., Bastoul, C., Cohen, A., Parello, D., Sigler, M., and Temam, O. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Program. 34* (June 2006), 261–317.

[20] Girbal, S., Vasilache, N., Bastoul, C., Cohen, A., Parello, D., Sigler, M., and Temam, O. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming 34*, 3 (June 2006), 261–317.

[21] Govindaraju, N. K., Larsen, S., Gray, J., and Manocha, D. A memory model for scientific algorithms on graphics processors. In *Supercomputing, 2006. SC '06. Proceedings of the ACM/IEEE SC 2006 Conference* (2006), p. 6.

[22] Hall, M., Anderson, J., Amarasinghe, S., Murphy, B., Liao, S., Bugnion, E., and Lam, M. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer 29*, 12 (Dec. 1996), 84–89.

[23] Hall, M., Chame, J., Shin, J., Chen, C., Rudy, G., and Khan, M. M. Loop transformation recipes for code generation and auto-tuning. In *LCPC* (October, 2009).

[24] Han, T. D., and Abdelrahman, T. S. hicuda: a high-level directive-based language for gpu programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units* (New York, NY, USA, 2009), GPGPU-2, ACM, pp. 52–

61.

[25] Hartono, A., Norris, B., and Sadayappan, P. Annotation-based empirical performance tuning using orio. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 1–11.

[26] Herrero, J. R., and Navarro, J. J. Improving performance of hypermatrix cholesky factorization. In *9th International Euro-Par Conference* (2003), pp. 461–469.

[27] Ierusalimschy, R., de Figueiredo, L. H., and Filho, W. C. Lua an extensible extension language. *Softw. Pract. Exper. 26* (June 1996), 635–652.

[28] Jiménez, M., Llabería, J. M., and Fernández, A. Register tiling in nonrectangular iteration spaces. *ACM Trans. Program. Lang. Syst. 24*, 4 (July 2002), 409–453.

[29] Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., and Wonnacott, D. The omega library interface guide. Tech. rep., College Park, MD, USA, 1995.

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

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

[32] Kisuki, T., Knijnenburg, P. M. W., and O'Boyle, M. F. P. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Oct. 2000).

[33] Knijnenburg, P. M. W., Kisuki, T., Gallivan, K., and O'Boyle, M. F. P. The effect of cache models on iterative compilation for combined tiling and unrolling. *Concurrency and Computation: Practice and Experience 16*, 2–3 (Mar. 2004), 247–270.

[34] Kodukula, I., Ahmed, N., and Pingali, K. Data-centric multi-level blocking. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 1997).

[35] Kuck, D. J., Kuhn, R. H., Padua, D. A., Leasure, B., and Wolfe, M. Dependence graphs and compiler optimizations. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'81)* (Jan. 1981).

[36] Lee, S., Min, S.-J., and Eigenmann, R. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2009), PPoPP '09, ACM, pp. 101–110.

[37] Lim, A. W., and Lam, M. S. Maximizing parallelism and minimizing synchronization with affine partitioning. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on*

Principles of Programming Languages (POPL'97) (Jan. 1997).

[38] Lim, A. W., Liao, S.-W., and Lam, M. S. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (June 2001).

[39] Lu, Q., Krishnamoorthy, S., and Sadaypppan, P. Combining analytical and empirical approaches in tuning matrix transposition. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Sept. 2006).

[40] McKinley, K. S., Carr, S., and Tseng, C.-W. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst. 18*, 4 (July 1996), 424–453.

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

[42] Olano, M., and Lastra, A. A shading language on graphics hardware: the pixelflow shading system. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1998), SIGGRAPH '98, ACM, pp. 159–168.

[43] Peercy, M. S., Olano, M., Airey, J., and Ungar, P. J. Interactive multi-pass programmable shading. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2000), SIGGRAPH '00, ACM Press/Addison-Wesley Publishing Co., pp. 425–432.

[44] Pouchet, L.-N., Bastoul, C., Cohen, A., and Cavazos, J. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *Proceedings of the International Symposium on Code Generation and Optimization* (Mar. 2007).

[45] Pouchet, L.-N., Bastoul, C., Cohen, A., and Vasilache, N. Iterative optimization in the polyhedral model: Part II, multi-dimensional time. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 2008).

[46] Proudfoot, K., Mark, W. R., Tzvetkov, S., and Hanrahan, P. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), SIGGRAPH '01, ACM, pp. 159–170.

[47] Pugh, W., and Rosser, E. Iteration space slicing for locality. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing* (Aug. 1999).

[48] Qasem, A., Jin, G., and Mellor-Crummey, J. Improving performance with integrated program transformations. Technical Report TR03-419, Rice University, Oct. 2003.

[49] Qasem, A., and Kennedy, K. Profitable loop fusion and tiling using model-driven empirical search. In *Proceedings of the 2006 ACM International Conference on Supercomputing* (June 2006).

[50] Ren, M., Park, J. Y., Houston, M., Aiken, A., and Dally, W. J. A tuning framework for software-managed memory hierarchies. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Oct. 2008).

[51] Rhoades, J., Turk, G., Bell, A., State, A., Neumann, U., and Varshney, A. Real-time procedural textures. In *Proceedings of the 1992 symposium on Interactive 3D graphics* (New York, NY, USA, 1992), I3D '92, ACM, pp. 95–100.

[52] Rivera, G., and Tseng, C.-W. Data transformations for eliminating conflict misses. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 1998).

[53] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W.-m. W. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (New York, NY, USA, 2008), PPoPP '08, ACM, pp. 73–82.

[54] Sarkar, V., and Thekkath, R. A general framework for iteration-reordering loop transformations. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 1992).

[55] Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., and Hanrahan, P. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph. 27* (August 2008), 18:1–18:15.

[56] Swaminarayan, S., Kadau, K., Germann, T. C., and Fossum, G. C. 369 tflop/s molecular dynamics simulations on the roadrunner general-purpose heterogeneous supercomputer. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (Piscataway, NJ, USA, 2008), SC '08, IEEE Press, pp. –10.

[57] Temam, O., Granston, E. D., and Jalby, W. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing '93* (Nov. 1993).

[58] Ueng, S.-Z., Lathara, M., Baghsorkhi, S. S., and mei W. Hwu, W. Cuda-lite: Reducing GPU programming complexity. In *LCPC* (2008), pp. 1–15.

[59] Volkov, V., and Demmel, J. W. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of Supercomputing '08* (Nov. 2008).

[60] Volkov, V., and Demmel, J. W. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (2008), IEEE Press, pp. 1–11.

[61] Whaley, R. C., and Whaley, D. B. Tuning high performance kernels through empirical compilation. In *Proceedings of the 34 International Conference on Parallel Processing* (June 2005).

[62] Wilson, R. P., French, R. S., Wilson, C. S., Amarasinghe, S. P., Anderson, J. M., Tjiang, S. W. K., Liao, S.-W., Tseng, C.-W., Hall, M. W., Lam, M. S., and Hennessy, J. L. Suif: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not. 29* (December 1994), 31–37.

[63] Wolf, M. E. *Improving Locality and Parallelism in Nested Loops.* PhD thesis, Stanford University, Aug. 1992.

[64] Wolf, M. E., and Lam, M. S. A data locality optimizing algorithm. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 1991).

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

[66] Wolf, M. E., Maydan, D. E., and Chen, D.-K. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture* (Dec. 1996).

[67] Wolfe, M. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing* (Dec. 1987).

[68] Wolfe, M. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 1989), Supercomputing '89, ACM, pp. 655–664.

[69] Wolfe, M. Data dependence and program restructuring. *The Journal of Supercomputing 4*, 4 (Jan. 1991), 321–344.

[70] Wolfe, M. Compilers and more: Optimizing GPU kernels. `http://www.hpcwire.com/features/Compilers_and_More_Optimizing_GPU_Kernels.html`, Oct. 2008.

[71] Yi, Q., Seymour, K., You, H., Vuduc, R., and Quinlan, D. Poet: Parameterized optimizations for empirical tuning. *In Proceedings of the 21st International Parallel and Distributed Processing Symposium* (Mar. 2007).