2017

An Optimization Compiler Framework Based on Polyhedron Model for GPGPUs

Lifeng Liu
Wright State University

Follow this and additional works at: https://corescholar.libraries.wright.edu/etd_all
Part of the Computer Engineering Commons, and the Computer Sciences Commons

Repository Citation
https://corescholar.libraries.wright.edu/etd_all/1746

This Dissertation is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact corescholar@www.libraries.wright.edu, library-corescholar@wright.edu.
AN OPTIMIZATION COMPILER FRAMEWORK BASED ON POLYHEDRON MODEL FOR GPGPUS

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

by

LIFENG LIU
B.E., Shanghai Jiaotong University, 2008
M.E., Shanghai Jiaotong University, 2011

2017
WRIGHT STATE UNIVERSITY

___________________________
Meilin Liu, Ph.D.
Dissertation Director

___________________________
Michael L. Raymer, Ph.D.
Director, Computer Science and Engineering Ph.D. program

___________________________
Robert E.W. Fyffe, Ph.D.
Vice President for Research and Dean of the Graduate School

Committee on Final Examination

___________________________
Meilin Liu, Ph.D.

___________________________
Jack Jean, Ph.D.

___________________________
Travis Doom, Ph.D.

___________________________
Jun Wang, Ph.D.
ABSTRACT


General purpose GPU (GPGPU) is an effective many-core architecture that can yield high throughput for many scientific applications with thread-level parallelism. However, several challenges still limit further performance improvements and make GPU programming challenging for programmers who lack the knowledge of GPU hardware architecture. In this dissertation, we describe an Optimization Compiler Framework Based on Polyhedron Model for GPGPUs to bridge the speed gap between the GPU cores and the off-chip memory and improve the overall performance of the GPU systems.

The optimization compiler framework includes a detailed data reuse analyzer based on the extended polyhedron model for GPU kernels, a compiler-assisted programmable warp scheduler, a compiler-assisted cooperative thread array (CTA) mapping scheme, a compiler-assisted software-managed cache optimization framework, and a compiler-assisted synchronization optimization framework. The extended polyhedron model is used to detect intra-warp data dependencies, cross-warp data dependencies, and to do data reuse analysis. The compiler-assisted programmable warp scheduler for GPGPUs takes advantage of the inter-warp data locality and intra-warp data locality simultaneously. The compiler-assisted CTA mapping scheme is designed to further improve the performance of the programmable warp scheduler by taking inter thread block data reuses into consideration. The compiler-assisted software-managed cache optimization framework is designed to make a better use of the shared memory of the GPU systems and bridge the speed gap between the GPU cores and global off-chip memory. The synchronization optimization framework is developed to automatically insert synchronization statements into GPU kernels at compile time, while simultaneously minimizing the number of inserted synchronization statements.

Experiments are designed and conducted to validate our optimization compiler frame-
work. Experimental results show that our optimization compiler framework could automatically optimize the GPU kernel programs and correspondingly improve the GPU system performance. Our compiler-assisted programmable warp scheduler could improve the performance of the input benchmark programs by 85.1% on average. Our compiler-assisted CTA mapping algorithm could improve the performance of the input benchmark programs by 23.3% on average. The compiler-assisted software managed cache optimization framework improves the performance of the input benchmark applications by 2.01x on average. Finally, the synchronization optimization framework can insert synchronization statements automatically into the GPU programs correctly. In addition, the number of synchronization statements in the optimized GPU kernels is reduced by 32.5%, and the number of synchronization statements executed is reduced by 28.2% on average by our synchronization optimization framework.
## Contents

1 Chapter 1: Introduction 1
1.1 Background ........................................ 1
1.2 The Challenges Of GPU Programming .................. 3
1.3 Our Approaches and Contributions .................... 5
  1.3.1 The Optimization Compiler Framework ............... 6
  1.3.2 The Compiler-assisted Programmable Warp Scheduler .... 7
  1.3.3 The Compiler-assisted CTA Mapping Scheme ........... 7
  1.3.4 The Compiler-assisted Software-managed Cache Optimization Framework ..................... 8
  1.3.5 The Compiler-assisted Synchronization Optimization Framework .... 8
  1.3.6 Implementation of Our Compiler Optimization Framework .... 8
1.4 Dissertation Layout .................................. 9

2 Chapter 2: Basic Concepts 11
2.1 The Hardware Architectures of GPGPUs ................. 11
  2.1.1 The Overview of GPGPUs .......................... 11
  2.1.2 Programming Model ............................... 12
  2.1.3 The CTA mapping ................................. 13
  2.1.4 The Basic Architecture Of A Single SM ............... 14
  2.1.5 The Memory System .............................. 18
  2.1.6 The Barrier Synchronizations ..................... 19
2.2 Basic Compiler Technologies .......................... 20
  2.2.1 Control Flow Graph ............................... 20
  2.2.2 The Dominance Based Analysis and the Static Single Assignment (SSA) Form ................. 21
  2.2.3 Polyhedron Model ................................ 24
  2.2.4 Data Dependency Analysis Based On The Polyhedron Model ..... 32

3 Chapter 3: Polyhedron Model For GPU Programs 37
3.1 Overview ........................................... 37
3.2 Preprocessor ......................................... 38
3.3 The Polyhedron Model for GPU Kernels ................ 41
# Chapter 7: Compiler-assisted Software-Managed Cache Optimization Framework

## 7.1 Introduction

### 7.1.1 Motivation

### 7.1.2 Case Study

## 7.2 Compiler-assisted Software-managed Cache Optimization Framework

### 7.2.1 An Illustration Example

### 7.2.2 The Mapping Relationship Between the Global Memory Accesses and the Shared Memory Accesses

### 7.2.3 The Data Reuses In the software-managed Cache

## 7.3 Compiler Supporting The Software-Managed Cache

### 7.3.1 Generate BASEs

### 7.3.2 Generate SIZEs

### 7.3.3 Validation Checking

### 7.3.4 Obtain The Best STEP Value

## 7.4 Evaluation

### 7.4.1 Experimental Platform

### 7.4.2 Experimental Results

## 7.5 Limitations

## 7.6 Related Work

## 7.7 Summary

---

## Chapter 8: Conclusions and Future Work

## 8.1 Conclusions

## 8.2 Future Work

---

**Bibliography**
List of Figures

1.1 The simplified memory hierarchy of CPUs and GPUs .............................. 3
1.2 Compiler optimization based on the polyhedron model for GPU programs .... 6

2.1 The basic architectures of GPGPUs [22] ............................................. 12
2.2 The SPMD execution model [22] ......................................................... 13
2.3 Two dimensional thread organization in a thread block and a thread grid [22] 14
2.4 The CTA mapping [40] ................................................................. 15
2.5 The basic architecture of a single SM [32] ............................................ 16
2.6 The memory system architecture [5] .................................................... 18
2.7 The overhead of barrier synchronizations [21] ..................................... 19
2.8 The pipeline execution with barrier synchronizations [5] ....................... 20
2.9 An example CFG [50] ................................................................. 21
2.10 The dominance relationship [50] ....................................................... 22
2.11 The IDOM tree [50] ................................................................. 23
2.12 Renaming variables [50] ............................................................... 25
2.13 Multiple reaching definitions [50] .................................................... 25
2.14 Merge function [50] ................................................................. 25
2.15 Loop nesting level ................................................................. 26
2.16 A statement enclosed in a two-level loop ........................................ 28
2.17 The example iteration domain for statement S1 in Figure 2.16 (N=5) ........ 28
2.18 An example code segment with a loop .............................................. 31
2.19 The AST for the code in Figure 2.18 .................................................. 31
2.20 An example code segment for data dependency analysis ...................... 34
2.21 Data dependency analysis (For N=5) [29] ......................................... 36

3.1 The general work flow of our compiler framework ............................... 38
3.2 Intermediate code ............................................................................ 40
3.3 Micro benchmark ............................................................................. 42

4.1 The memory access pattern for matrix multiplication with the round-robin warp scheduler, in which i represents the loop iteration .................... 49
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The dominance relationship analysis</td>
<td>23</td>
</tr>
<tr>
<td>4.1</td>
<td>The baseline simulator configuration</td>
<td>76</td>
</tr>
<tr>
<td>4.2</td>
<td>The benchmarks used in the evaluation [15]</td>
<td>76</td>
</tr>
<tr>
<td>4.3</td>
<td>The estimation accuracy</td>
<td>78</td>
</tr>
<tr>
<td>6.1</td>
<td>Comparisons of the number of synchronizations</td>
<td>131</td>
</tr>
<tr>
<td>7.1</td>
<td>Hardware platform</td>
<td>168</td>
</tr>
<tr>
<td>7.2</td>
<td>STEP value and cache size for the 16k and 48k software cache configuration</td>
<td>169</td>
</tr>
</tbody>
</table>
Acknowledgment

I would like to extend my thanks to my adviser and Ph.D. dissertation supervisor, Dr. Meilin Liu, for her motivation, unconditional encouragement, support and guidance. Especially I would like to thank her patience and the long hour discussions of the research problems presented in this dissertation.

I would also like to thank Dr. Jun Wang, Dr. Jack Jean and Dr. Travis Doom for reviewing this dissertation and serving on my Ph.D. dissertation committee. Thanks for their valuable suggestions and firm supports.

Next, I would like to give my thanks to Dr. Mateen Rizki, the chair of the Department of Computer Science and Engineering, and the stuff of the department of Computer Science and Engineering for their help.

Finally, I would like to thank my colleagues, my family and my friends. Their supports and encouragements help me overcome the difficulties I have encountered during the research of this dissertation.
Dedicated to

my wife Kexin Li
Chapter 1: Introduction

1.1 Background

As the power consumption and chip cooling technology are limiting the frequency increase of the single core CPUs, multi-core and many-core processors have become the major trend of computer systems[1, 22, 35, 58, 17]. General purpose GPU (GPGPU) is an effective many-core architecture for computation intensive applications both in scientific research and everyday life [22, 52]. Compared to the traditional CPUs such as Intel X86 serial CPUs, GPGPUs have significant advantages for certain applications [22, 52].

First, the single chip computation horsepower of GPGPUs is much higher than the traditional CPUs. As reported by Nvidia [22], the single precision GFlops of GeForce 780 Ti GPU chip is higher than 5000, which is more than 10 times higher than the top Intel CPUs. In addition, the double precision GFlops of Tesla K40 GPU chip reaches nearly 1500, which is also nearly 10 times compared to the top Intel CPUs. GPUs achieve the high computation power by putting hundreds or even thousands of parallel stream processor cores into one chip. Compared to the CPUs which have very strong single thread processing power, the computation speed of a single thread on the GPU systems is very modest. However, the massively parallel structure of GPUs, which enables thousands of threads work in parallel, improves the overall computation power.

Second, the memory access throughput of the GPUs is much higher than the traditional CPUs. The memory bandwidth of GeForce 780 Ti GPU and Tesla K40 GPU is
nearly 6 times higher than the top Intel CPUs. Compared to the processing core, DRAMs (Dynamic random-access memory) have much lower clock speed, and the latency between the memory access requests and responses is also very high. For memory, and IO intensive applications the memory wall problem becomes the bottleneck that limits the overall system performance. Higher memory bandwidth will leverage the overall performance of those applications.

Third, compared to the supercomputers consisting of multiple CPU chips, GPUs could achieve the same computation power with lower power consumption and economic cost. Supercomputer servers usually operate on large frames, which are usually very expensive and need professional management. Generally speaking, supercomputers are only affordable for large companies and research institutes. However, GPUs can be installed in regular PCs, which makes massively parallel computing more feasible. And at the same time, the size of the computers is reduced significantly. For example, medical image systems such as X-ray computed tomography (X-ray CT) and Magnetic resonance imaging (MRI) usually need large amount of computations to construct the images. Computer systems based on the CPUs for MRI have limited portability and the cost of those systems is high. GPUs can make those computer systems much smaller and easier to use in a clinical setting.

Generally speaking, GPUs provide significant computation speed improvement for parallel applications as the co-processor of traditional CPUs. In addition, the development of GPU programming interfaces such as CUDA and OpenCL makes the programming on GPUs much easier [22, 17]. More and more developers have ported their applications from the traditional CPU based platforms to GPU platforms [12, 25, 42, 16]. And the GPU platforms have already become a research hot spot [45, 5, 33, 32].
1.2 The Challenges Of GPU Programming

GPGPU is an effective many-core architecture for scientific computations by putting hundreds or even thousands stream processor cores into one chip. However, there are still several challenges for GPU programming [22, 52].

Limited by the current DRAM technologies, accessing off-chip memory is very time consuming. In traditional CPUs, large L1 and L2 caches play an important role in bridging the speed gap between CPU cores and the off-chip DRAM memory. Both L1 and L2 caches are managed automatically by the hardware and are transparent to programmers. Programmers only need to consider a uniform continuous memory space. However, the memory hierarchy on GPU platforms is different (as illustrated in Figure 1.1). On each SM core of a GPU chip, a small high-speed software-managed scratch pad memory called shared memory is designed to cache small scale of frequently used data and the data shared among different threads in the same thread block. The first challenge to use shared memory is that the shared memory locates in a different memory space outside the main global memory of GPUs. To bring data into the shared memory of GPU systems, the GPU programmers must specifically copy the data from the global memory to the shared memory manually. Second,
to take advantage of the shared memory, the GPU programmers must have the hardware knowledge of GPUs, which makes GPU programming more challenging. To use the shared memory effectively, the GPU programmers have to apply data reuse analysis before they make decision on how to buffer the data in the shared memory. Only the data blocks that are indeed reused during the execution should be brought into the shared memory. Finally, limited by the small size of the shared memory, too much shared memory usage will reduce the overall performance of the GPU systems substantially. So GPU programmers must tune their program and arrange the data buffered in the shared memory properly to achieve the best performance. On the other hand, when the shared memory is used, barrier synchronizations must be inserted to preserve the data dependencies, so that the GPU programs’ semantics are preserved. The data dependency analysis also needs expert knowledge and deep understanding of GPU hardware.

In addition to the shared memory, GPUs do have L1 and L2 caches as shown in Figure 1.1, however the competitions for cache resources among different threads are more intensive on the GPU platforms. The number of threads concurrently executed on the CPU chips is usually comparable to the number of cores, so the cache resources in CPU systems are usually shared by a small number of parallel threads. However, in a GPU SM core, hundreds or even thousands of concurrent threads are executed simultaneously, all of which are sharing the same L1 cache (The size of the L1 cache of each SM is usually 16KB, which is smaller than the L1 data cache for most CPU cores). The L2 caches are shared among different SM cores, and the competitions for L2 caches are also intensive.

Caches only improve the overall system performance when the data blocks cached are reused by later memory accesses. When too many threads are competing for the same cache block, the data blocks in the cache that can be reused later might be evicted before they are reused. Such unnecessary cache block evictions can lead to cache thrashing. Cache thrashing increases the number of global memory accesses and degrades the overall system performance [51, 33, 47]. Unfortunately, the default warp scheduler and CTA (Cooperative
Thread Array) mapping module in current GPU hardware do not consider the data locality during the execution of parallel GPU programs and are prone to issue too many concurrent threads simultaneously, which makes the situation even worse.

Several optimized warp scheduling algorithms have been proposed to alleviate the problem [33, 47, 51]. However, without a detailed data reuse analysis framework, their solutions either need manually profiling, which increases the workloads of GPU programmers, or need extra complex hardware support which has very large overhead.

1.3 Our Approaches and Contributions

As stated in Section 1.2, there are several challenges making GPU programming challenging for regular programmers who do not have the hardware knowledge of the GPUs and preventing further performance improvements of GPU systems. Generally speaking, porting a program to the GPU platform is not difficult. However, optimizing the GPU programs to achieve the best performance requires the knowledge of the GPU hardware. In this dissertation, an Optimization Compiler Framework Based on Polyhedron Model for GPGPUs is designed to bridge the speed gap between the GPU core and the off-chip memory to improve the overall performance of GPU systems. The optimization compiler framework includes a detailed data reuse analyzer based on the extended polyhedron model for GPU kernels, a compiler-assisted programmable warp scheduler, a compiler-assisted CTA mapping scheme, a compiler-assisted software-managed cache optimization framework, and a compiler-assisted synchronization optimization framework to help the GPU programmers optimize their GPU kernels.

The core part of the optimization compiler framework is a detailed data reuse analyzer of GPU programs. The extended polyhedron model is used to detect intra-warp data dependencies, cross-warp data dependencies, and to do data reuse analysis. Compared to traditional data dependency analysis technologies such as distance vectors [50], polyhe-
dron model have several advantages. First, polyhedron model can handle perfectly nested loops and imperfectly nested loops simultaneously, which is more general compared to the traditional data dependency analysis technologies. Second, based on the polyhedron model, the parameters related to a data dependency such as data reuse distance are much easier to obtain through a rigorous mathematical model. Third, polyhedron model is more friendly for high-level program analysis. The polyhedron model can be generated from the source code directly and the output source code can be recovered based on the polyhedron model of a program.

1.3.1 The Optimization Compiler Framework

The overall architecture of our optimization compiler framework is illustrated in Figure 1.2. As shown in Figure 1.2, our first contribution is that we extend the traditional polyhedron model for CPU programs to the polyhedron model for GPU programs. Compared to the CPU programs, the execution domain of each statement of a GPU program is represented in the extended polyhedron model to consider the parallel threads in the GPU programs. In addition, the memory accesses on the multiple levels of the GPU memory hierarchy also need to be represented systematically. So the memory hierarchy information is also needed in the extended polyhedron model for GPU programs. To make the GPU programs ready to be converted to the polyhedron model representation, we design a preprocessor to parse the input GPU kernels to collect the information needed. We design different types of
reuse analyzer in our optimization compiler framework, based on the extended polyhedron model, which is a powerful and flexible tool.

1.3.2 The Compiler-assisted Programmable Warp Scheduler

The second contribution is that we design a compiler-assisted programmable warp scheduler that can reduce the performance degradation caused by L1 cache thrashing. By constructing a long term high priority warp group, some warps gain advantage in the competition for cache resources. With less number of the threads competing for the L1 cache, the number of unnecessary cache evictions will be reduced. The high priority warp groups will be dynamically created and destroyed by special software instructions. Compared to the hardware controlled warp scheduler such as CCWS [51], the overhead of our programmable warp scheduler is much smaller. The programmable warp scheduler works for the applications that do have intra-warp or inter-warp data reuses. However, the size of the high priority warp group also affects the overall system performance. So the optimization compiler framework supporting the programmable warp scheduler has a data reuse analyzer to detect both intra-warp and inter-warp data reuses to help the programmable warp scheduler decide whether or not the high priority warp group is needed. In addition, a cache boundary detector based on the data reuse analyzer for concurrent memory accesses is designed to determine the best warp group size to avoid self-evictions, i.e., the unnecessary cache evictions that degrade the overall performance substantially.

1.3.3 The Compiler-assisted CTA Mapping Scheme

The third contribution is that we design a compiler-assisted CTA mapping scheme to preserve the inter thread block data locality. Inter thread block data reuse analyzer is used to detect the data reuse patterns among different thread blocks. The compiler-assisted CTA mapping scheme could be combined with our programmable warp scheduler to further
1.3.4 The Compiler-assisted Software-managed Cache Optimization Framework

Our fourth contribution is the design of the compiler-assisted software-managed cache optimization framework, which can help the GPU programmers use the shared memory more effectively and optimize the performance of the shared memory automatically. The parameters needed for optimizing the software-managed cache is obtained based on the detailed data reuse analysis based on the polyhedron model.

1.3.5 The Compiler-assisted Synchronization Optimization Framework

The last contribution is the design of the compiler-assisted synchronization optimization framework. The compiler-assisted synchronization optimization framework automatically inserts the synchronization statements into GPU kernels at compile time, while simultaneously minimize the number of inserted synchronization statements. In GPU programs based on the lock step execution model, only cross warp data dependencies (CWD) need to be preserved by the barrier synchronizations. The data reuse analyzer is used to detect whether or not a data dependency is CWD to avoid unnecessary barrier synchronizations.

1.3.6 Implementation of Our Compiler Optimization Framework

The optimization compiler framework is implemented based on the extended CETUS environment [36] and evaluated on the GPU platforms, or the GPU simulator, GPGPU-sim. Experimental results show that our optimization compiler framework can automatically optimize the GPU kernel programs and correspondingly improve the performance.

The optimization compiler framework has a preprocessor and a post-processor. The
preprocessor and the post-processor are based on the enhanced CETUS compiler framework proposed by Yang et al. [61, 36]. The preprocessor is used to preprocess the input GPU kernels to generate the intermediate code and gather information needed for data reuse analysis based on the polyhedron model. The polyhedron model is generated by a modified polyhedron compiler front end “clan” [8]. Based on the polyhedron model we can do data reuse analysis. The optimization (such as loop tiling) can be applied based on the result of data reuse analysis. The polyhedron model will be transferred to the intermediate code by a modified polyhedron scanner “cloog” [13]. Finally, the optimized output GPU kernel is recovered by a post-processor based on the output intermediate code and data reuse analysis with the optimization applied.

1.4 Dissertation Layout

The rest of this dissertation is organized as follows: in Chapter 2, we introduce the basic concepts related to GPU hardware architecture and the basic compiler technologies used in this dissertation.

In Chapter 3, we introduce the extended polyhedron model for GPU kernels, the preprocessor of the GPU programs, and how to obtain the extended polyhedron model parameters for GPU programs.

In chapter 4, we first illustrate how the polyhedral model for GPU kernels is used to detect intra-warp and inter-warp data reuses of a GPU kernel. The scheduling algorithms and the implementation details of the compiler-assisted programmable warp scheduler are then presented. The experimental results are subsequently presented.

In chapter 5, we describe a compiler-assisted CTA mapping scheme that can detect and take advantage of the inter thread block data reuses. We discuss how the polyhedron model for GPU kernels is used to detect inter thread block data reuses. We design new CUDA runtime APIs to facilitate the implementation of the compiler-assisted CTA mapping scheme.
Experimental results are reported to validate the compiler-assisted CTA mapping scheme.

In chapter 6, we discuss data dependence analysis and derive the basic rules of synchronization insertion. We then elaborate on the design of our compiler-assisted synchronization optimization framework. We subsequently present our experimental results.

In Chapter 7, we illustrate how the source-to-source compiler framework uses the data reuse analyzer to generate the parameters for the software-managed cache, how to identify the global memory accesses that can take advantage of the shared memory buffers, and how to transform the global memory accesses to shared memory accesses automatically. We then discuss the design of the compiler-assisted synchronization optimization framework. Finally, we present the experimental results.

We conclude the dissertation and discuss our future work in Chapter 8.
Chapter 2: Basic Concepts

In this chapter we present the basic concepts related to the technologies presented in this dissertation. In Section 2.1, we introduce the basic hardware architectures of GPGPUs. In Section 2.2, we present basic concepts related to our compiler framework.

2.1 The Hardware Architectures of GPGPUs

2.1.1 The Overview of GPGPUs

As illustrated in Figure 2.1, GPGPUs [22, 21, 9] consist of several Streaming Multiprocessors (SMs), each of which has multiple streaming processors (SPs). The GPU kernel programs and the data are transferred into the global memory of the GPUs by the host CPU through the PCI-e bus. The global memory usually uses the off-chip Double Data Rate (GDDR) DRAMs. The capacity could be as large as tens of gigabytes. In order to achieve the high global memory access bandwidth, several memory access controllers are introduced, each of which has an on-chip L2 data cache. Those SMs and memory access controllers are connected together by the on-chip interconnection network. The parallel tasks loaded on a GPU system will be managed by the execution manager.
2.1.2 Programming Model

The Compute Unified Device Architecture (CUDA) programming model [22, 21] introduced by Nvidia is the basic programming model for Nvidia GPUs. CUDA extends the normal C programming languages to support multi-thread execution on the GPUs. The programs executed on the GPUs are defined as kernel functions. Kernel programs will be executed in single program multiple data (SPMD) manner. All of the parallel threads perform the same sequence of calculations on different pieces of data.

In the GPU program shown in Figure 2.2(a), a CUDA kernel function `add()` is launched in the `main()` function. When the kernel is executed on the GPU, \( N \) threads will be launched. Those threads will be distinguished by the unique thread ID of each thread. During the execution, the special key word `threadIdx.x` in the GPU kernel program will be replaced by the hardware defined thread ID. Then each thread can perform operations in parallel on different pieces of data decided by the thread ID as shown in Figure 2.2(b).

To facilitate the task management, the CUDA program model can organize the grid
of threads in multi-dimensions [22, 21, 18]. An example thread grid is illustrated in Figure 2.3. The threads in a thread block can be indexed by 1-dimensional, 2-dimensional or 3-dimensional vectors based on the thread organization. The size of each dimension is limited by the hardware. For example, Nvidia Fermi and Kepler GPUs both limit the maximum size of the thread block as 1024 in each dimension. Each thread block is also called a CTA since all the threads in each thread block will be assigned to an SM. On the other hand, the threads in the same thread block can collaborate with each other through barrier synchronization statements and communicate with each other through the shared memory. The thread blocks in a thread grid are also organized in multi-dimensions as shown in Figure 2.3. A thread block in a thread grid can be indexed by a 1-dimensional or 2-dimensional vector. The maximum sizes along each dimension of the thread grid are also limited by the hardware, usually 65536.

2.1.3 The CTA mapping

Threads belonging to different thread blocks (CTAs) can be executed independently. Multiple CTAs can be assigned and executed concurrently on one SM. During the execution of a GPU kernel, the maximum number of CTAs that can be mapped to a SM is limited by
the hardware resources occupied by each CTA and the hardware parameters. The relevant hardware resources are the register usage and the shared memory usage. The maximum number of CTAs mapped to an SM can be calculated as follows [22, 21]:

\[
#CTAs = \min\left(\frac{\text{total\#registers}}{\text{registersPerCTA}}, \frac{\text{totalSharedMemory}}{\text{SharedMemoryPerCTA}}, \text{#hardwareCTASlots}\right)
\]  

(2.1)

For Nvidia Fermi and Kepler GPUs, the maximum number of hardware CTA slots is eight [19]. New thread blocks will be assigned to an SM when a thread block on this SM terminates. The default thread block is selected in a round-robin manner [40, 22] as illustrated in Figure 2.4.

2.1.4 The Basic Architecture Of A Single SM

A single SM is a single instruction multiple data (SIMD) core [22, 32, 5]. The SIMD width for Fermi and Kepler GPUs is 32. So 32 ALUs work simultaneously on 32 different threads. The thread blocks assigned to an SM are divided into fixed-size thread groups called warps. The warp size is equal to the SIMD width. Usually there are 32 threads in
a warp for Nvidia GPUs. The warps ready for execution will be put in a ready queue. A scheduler will select one of the ready warps for execution in each clock cycle.

The pipeline of an SM can be briefly divided into 5 stages [5]: instruction fetch, instruction decode, register access, execute, and write back. The threads in the same warp will share the instruction fetch, instruction decode and write back stage and they share the same program counter. So all of the threads in the same warp execute one common instruction in lock step manner at a time. At each clock cycle, the instruction fetch unit will fetch the next instruction of all the threads in the warp with the same address concurrently. To support the concurrent execution of those parallel threads, SMs usually have very large register files (For example, Fermi GPUs have 128KB of registers per SM [19] and Kepler GPUs have 256KB of registers per SM [20]). Those registers will be divided evenly among the threads assigned to an SM. Each thread will be executed on it’s own ALU and will have its own register file section. Each thread corresponds to one stand-alone ALU, so all the ALUs will execute in parallel.

After the execution stage is finished, if the instruction being executed is a memory ac-
Figure 2.5: The basic architecture of a single SM [32]
cess instruction, then the memory access unit will read or write the corresponding memory address, perform register write back, and finish the execution of this instruction. Otherwise, the result of the calculation will be written back to the register file. The memory access unit works independently with ALUs, which will help the GPUs hide long latency operations. When a warp is waiting for the data located in the global memory, other warps can do ALU operations at the same time [5].

Usually, accessing off chip DRAM memory is very time consuming, and it may take tens or hundreds of processor clock cycles. The on-chip high-speed L1 data cache is used to speed up global memory accesses. If all the threads in a warp access data in the same cache line, the access operation will be performed in a single clock cycle. If the threads in a warp access different cache lines, the access operation will be serialized and one more clock cycle will be needed for each extra cache line access. If some of the threads in the warp access data blocks that have not been cached, the warp will be stalled and put into a waiting queue, waiting for the global memory access operation to be finished. So coalesced global memory accesses (threads in a warp access consecutive addresses) improve the global memory access performance significantly (32 times faster than the worst case) [32].

The waiting queue [5, 21] uses a FIFO strategy for global memory accesses. The first warp in the queue will be served first. If the threads in the warp are accessing different data blocks in the global memory, the access operation will also be serialized. So in the worst case, $N$ memory blocks will be brought into the cache, which will take $N \times t$ clock cycles, where $t$ is the time required for a single memory block transfer. After all the memory access requests have been processed in the warp, the warp will be re-enqueued into the ready queue.

GPGPU cores also provide a small on-chip high-speed shared memory, which can be used by the GPU programmers as a high-speed buffer manually [22, 18, 9]. All the threads assigned to the same core can access the data in the shared memory of this core. In order to handle conditional branches [32, 22, 9], GPGPUs use thread masks for flow
control. If some of the threads in a warp take one branch and others take another branch, both branches will be executed sequentially. The execution results of the threads not taking a given branch will not take effect using the masks in the warp. Branch re-convergence mechanisms are used in order to save clock cycles [32].

2.1.5 The Memory System

When a memory access request is issued by an SM, it first checks whether or not the required data is in the L1 data cache [5]. If the memory access is missed in the L1 data cache, then the missing status hold register (MSHR) is checked see if the memory request can be merged with the former missed global memory accesses. The MSHR has several entries. Each entry has a fully associated tag array that stores a tag of a cache block. A queue storing information of different memory accesses is linked to each entry in MSHR. Each queue stores the missed memory requests that can be merged [46]. The total number
of tags that can be stored in the MSHR and the number of memory request that can be merged for each tag is limited. So when the MSHR is not able to find a corresponding entry with the same tag of a memory request, it puts that memory request into the missing queue. Each clock cycle, the missing queue issues a global memory access request onto the interconnection network. When a global memory access is fulfilled, the returned result will be put into the response queue. Each clock cycle, the MSHR fulfills a missed memory request stored in the MSHR corresponding to the tag value of the head of the response queue. If all of the merged requests related to an entry are fulfilled, that entry is freed for new missed memory accesses.

2.1.6 The Barrier Synchronizations

In GPU programs, the shared memory is usually used to enable the threads in the same thread block share data with each other and collaborate with each other [21, 9, 22]. The barrier synchronizations are needed in the GPU kernels to preserve the data dependencies because the parallel execution order of the threads is not determined. When a thread reaches the synchronization statement, it will be blocked until all the threads in the same thread block reach the statement. In Nvidias CUDA GPGPU framework, the barrier synchronization is performed by inserting __syncthreads() statement in the GPU kernel programs [9, 18].
Figure 2.8: The pipeline execution with barrier synchronizations [5]

Figure 2.7 shows the overhead introduced by the barrier synchronizations. Assume there is a thread block with two warps, and the shadowed parts represent the overhead of the warp switching. The warp switching can lead to pipeline flushing as shown in Figure 2.8.

In Figure 2.8, we assume the execution pipeline has 4 stages and instruction 2 is a synchronization statement. Assume only after stage 4 the processor can know the execution result of an instruction. As shown in Figure 2.8, warp switching caused by barrier synchronizations will introduce three pipeline bubbles. The more the pipeline stages the larger the overhead is. So we need to keep the number of barrier synchronizations to a minimum.

2.2 Basic Compiler Technologies

2.2.1 Control Flow Graph

Control flow analysis [39, 50, 3] is a widely used technique in compiler optimization field. A control flow graph (CFG) can be used to model the control flow of a program [50]. Statements in a program are first divided into basic blocks which is defined as

Definition 1. Basic Block [50, 3] A basic block is a straight-line code sequence with no
branches in except to the entry and no branches out except at the exit. Two statements are in the same basic block if and only if the execution of an instruction in the block guarantees the execution can only proceed to the next statement.

A CFG $G = (V, E, s, e)$ is a node-weighted, edge-weighted directed graph, where $V$ is the set of the basic blocks in the program. $E \subseteq V \times V$ is the set of edges representing the execution paths between basic blocks. $s$ and $e$ represent the entry point and the exit point of the program respectively.

Figure 2.9 shows the example code segment and its corresponding CFG [50, 39]. The code segment can be divided into 6 basic blocks. The test clauses of the loop and the if statement are also treated as basic blocks. In this dissertation, the set of nodes that are the direct predecessors and successors of a node $X$ in CFG is denoted as $PRED(X)$ and $SUCC(X)$. For example in Figure 2.9, $PRED(1) = \{0, 5\}$, and $SUCC(2) = \{3, 4\}$.

### 2.2.2 The Dominance Based Analysis and the Static Single Assignment (SSA) Form

Several useful dominance based program analysis on a CFG [50, 3, 24] can be performed. In a CFG, node $X$ dominating node $Y$ ($\text{DOM}_Y X$) is defined as
Definition 2. $X DOM Y$ [50, 3] In a CFG if all paths from the entry to node $Y$ include node $X$, then node $X$ dominates node $Y$ ($X DOM Y$).

As illustrated in Figure 2.10, when $X DOM Y$, all paths from the entry point to node $Y$ can be broken into two parts $Entry \rightarrow X$ and $X \rightarrow Y$. The set of nodes in a CFG dominated by $X$ can be represented as $DOM^{-1}X$. Based on the dominate relationship, we can define the strict dominate relationship ($X SDOM Y$) in a CFG:

Definition 3. $X SDOM Y$ [50, 3] $X SDOM Y$ iff $X DOM Y$ and $X ! = Y$

and the immediate dominator of node $X$ can be defined as:

Definition 4. immediate dominator [50, 3] The immediate dominator of node $X$ is the closest strict dominator of node $X$.

The dominance relationship analysis for the CFG shown in Figure 2.9 is illustrated in Table 2.1. Immediate dominators can be used to construct the IDOM tree which indicates
Table 2.1: The dominance relationship analysis

<table>
<thead>
<tr>
<th>X</th>
<th>DOM(X)</th>
<th>SDOM(X)</th>
<th>DOM⁻¹(X)</th>
<th>IDOM(X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0}</td>
<td>{}</td>
<td>{0,1,2,3,4,5,6}</td>
<td>{}</td>
</tr>
<tr>
<td>1</td>
<td>{0,1}</td>
<td>{0}</td>
<td>{1,2,3,4,5,6}</td>
<td>{}</td>
</tr>
<tr>
<td>2</td>
<td>{0,1,2}</td>
<td>{0,1}</td>
<td>{2,3,4,5}</td>
<td>{1}</td>
</tr>
<tr>
<td>3</td>
<td>{0,1,2,3}</td>
<td>{0,1,2}</td>
<td>{3}</td>
<td>{2}</td>
</tr>
<tr>
<td>4</td>
<td>{0,1,2,4}</td>
<td>{0,1,2}</td>
<td>{4}</td>
<td>{2}</td>
</tr>
<tr>
<td>5</td>
<td>{0,1,2,5}</td>
<td>{0,1,2}</td>
<td>{5}</td>
<td>{2}</td>
</tr>
<tr>
<td>6</td>
<td>{0,1,6}</td>
<td>{0,1}</td>
<td>{6}</td>
<td>{1}</td>
</tr>
</tbody>
</table>

Figure 2.11: The IDOM tree [50]
the dominance relationship among all of the nodes in a CFG. The IDOM tree for the CFG in Figure 2.9 is shown in Figure 2.11.

The dominance frontiers (DF) can be used to locate the merge points of different execution paths. The dominance frontiers of a node \( X \) (\( DF(X) \)) is the set of nodes \( N \) such that \( X \) dominates some predecessors of \( N \), but not all, formally, \( DF(X) = SUCC(DOM^{-1}(X)) - SDOM^{-1}(X) \). The dominance frontiers (DF) and the immediate dominator (IDOM) of a node \( X \) can be computed with the algorithm illustrated in [50].

Basic dominance analysis of a program can be represented in the static single assignment (SSA) form [50, 3, 23, 24], which is widely used in modern compiler optimization techniques. In a program that adopts the SSA format, each variable has only one static assignment, which can make it easier to reason about values instead of variables. A program not in SSA form can be translated into SSA form with two steps [50]: (1) Rename each variable assignment with a unique name. (2) Rename all uses reached by those assignments accordingly. The conversion procedure is illustrated in Figure 2.12. In order to handle the multiple reaching definitions as illustrated in Figure 2.13, we insert merge function (\( \varphi \) function) to the merge node as illustrated in Figure 2.14. In the CFG, merge functions are inserted on dominance frontier nodes [50, 24].

### 2.2.3 Polyhedron Model

**Definition 5.** *Affine function* [10, 57] A function \( f(x_1, x_2, \ldots, x_k) \) is an affine function if it can be expressed in the form of

\[
f(x_1, x_2, \ldots, x_k) = c_0 + c_1 * x_1 + c_2 * x_2 + \ldots + c_k * x_k
\]  

(2.2)
Figure 2.12: Renaming variables [50]

Figure 2.13: Multiple reaching definitions [50]

Figure 2.14: Merge function [50]
Definition 6. **Loop nesting level** [29] In a multiple level loop, the loop nesting level $p$ indicates the number of loops nested around a loop or a statement as illustrated in Figure 2.15. The statements in Figure 2.15 has a loop level of 3.

Definition 7. **Iteration vector** [29, 10] Given a nested $n$-level loop, the iteration vector $\vec{i}$ of a particular iteration of the statements in the innermost loop is a vector of integers that contains the iteration numbers for each of the loops in order of the nesting level. Thus, the iteration vector is:

$$\vec{i} = (i_1, i_2, ..., i_n)^T$$  \hspace{1cm} (2.3)

Where $i_k, 1 \geq k \geq n$ represents the iteration number for the loop at loop nesting level $p = k - 1$. In the context of compiler optimization, $i_k$ is an integer.

Definition 8. **Affine hyperplane** [10] In an $n$-dimensional space a hyperplane can be defined as

$$c_0 + c_1 * x_1 + c_2 * x_2 + ... + c_n * x_n = 0$$  \hspace{1cm} (2.4)

Where $\vec{v} = (x_1, x_2, ..., x_n)$ is a vector in the space. $c_0, c_1, ..., c_k$ are constant coefficients. An affine hyperplane can be formally expressed as $\vec{c} * \vec{v}^T + c_0 = 0$. 

---

Figure 2.15: Loop nesting level

Where $c_0, c_1, c_2, ..., c_k$ are constant coefficients.
An affine hyperplane can divide the entire space into two half spaces which can be represented as $\vec{c} \ast \vec{v}^T + c_0 \geq 0$ and $\vec{c} \ast \vec{v}^T + c_0 \leq 0$.

**Definition 9. Polyhedron (Polytope)** [29, 10, 8] In an $n$-dimensional space a polyhedron can be defined as the intersection of a set of half spaces, which can be represented as a set of inequalities

$$
\begin{align*}
\vec{c}_1 \ast \vec{v}^T + c_{1,0} & \geq 0 \\
\vec{c}_2 \ast \vec{v}^T + c_{2,0} & \geq 0 \\
\vdots \\
\vec{c}_m \ast \vec{v}^T + c_{m,0} & \geq 0
\end{align*}
$$

(2.5)

In the context of loop analysis, loop iteration vectors are always vectors of integers. So in this dissertation the polyhedron refers to the set of the points with integer indexes inside the intersection of those half spaces.

**Definition 10. Iteration domain** [29] For a statement nested in an $n$-level loop, the set consisting of all of the possible iteration vectors for which that the statement can be executed is defined as the iteration domain of that statement. Iteration domain can be represented by a polyhedron bounded by the loop boundaries (The minimum and maximum value of the loop iteration vectors) of all the nested loops. Each possible iteration vector represents a dynamic instance of the statement. In this dissertation the iteration domain of a statement $S$ is usually represented with $D_S$.

The statement $S_1$ shown in Figure 2.16 is enclosed by two loops, and the iteration domain of $S_1$ can be represented as
01: for(int i=1;i<=N;i++)
02:   for(int j=1;j<=i;j++)
03:     S1;

Figure 2.16: A statement enclosed in a two-level loop

![Diagram](image)

Figure 2.17: The example iteration domain for statement S1 in Figure 2.16 (N=5)

\[
\begin{align*}
\begin{cases}
i \geq 1 \\
i \leq N \\
j \geq 1 \\
j - i \leq 0
\end{cases}
\end{align*}
\]  

(2.6)

The polyhedron [29] defined by the iteration domain is illustrated in Figure 2.17(a).

To be represented by an iteration domain, the loop boundaries must be affine functions of outer loop iteration vectors and program parameters. (This type of programs are also known as static control programs [29]). The upper bound of the \(i\) loop in Figure 2.16 is the program parameter \(N\). The upper bound of the \(j\) loop is an affine function of the outer loop index \(i\).

A dynamic instance of statement \(s\) can be denoted as \((s, \vec{a})\), where \(\vec{a}\) is the loop it-
eration vector of the instance. The precedence relationship between two instances can be represented by the polyhedron model. When an instance of statement \( s \) with iteration vector \( \vec{a} \) executes before the instance of statement \( t \) with iteration vector \( \vec{b} \). The relationship can be represented by

\[
(s, \vec{a}) \prec (t, \vec{b})
\]  

(2.7)

Where \( s \) and \( t \) can be the same statement or different statements. With \( N_{s,t} \) representing the number of common loops enclosing \( s \) and \( t \), the precedence relationship \([29, 10]\) can be represented as

\[
\begin{cases}
  a[0, ..., p-1] = b[0, ..., p-1] \land a[p] < b[p] & \text{when } 0 \leq p < N_{s,t} \\
  \vec{a} = \vec{b} \land T_{s,t} = true & \text{when } p = N_{s,t}
\end{cases}
\]  

(2.8)

Where \( p \) is the loop level. \( T_{s,t} \) is true when \( s \) is before \( t \) in the source code. When \( s \) and \( t \) are the same statement, \( T_{s,t} \) will be set to false.

The memory array access subscripts for statement \( S \) can be formally represented as a vector

\[
\vec{m}_s = (\alpha_0, \alpha_1, ..., \alpha_{m-1})
\]  

(2.9)

Each subscript of a array access can also be represented based on the polyhedron model when it can be represented by an affine function of the loop iteration vectors, in which the \( k^{th} \) subscript can be represented as \([29, 10]\)

\[
\alpha_k = c_0 * i_0 + c_1 * i_1 + \cdots + c_n * i_n
\]  

(2.10)

Where \( c_0, c_1, ..., c_n \) are constant coefficients. Array subscripts that cannot be represented as affine functions of the loop iteration vectors such as indirect memory accesses like \( a[b[i]] \) cannot be processed by the polyhedron model.
Definition 11. **Schedule** [30, 31, 8, 13] During the execution of a program, each instance of a statement can be assigned a unique logical time stamp. We call the collection of the time stamps for all the instances of all the statements as a schedule. In other words, the schedule is the mapping of all the statements to specific time stamps when the resource constraints, the data dependencies, and the control dependencies are satisfied. The schedule of a statement can be represented by a function of the corresponding iteration vector ($\vec{a}$):

\[ \text{schedule} = \theta(\vec{a}) \] (2.11)

When the real execution order complies with the logic time stamp, i.e., the instance executed earlier always has a smaller time stamp [30, 31], the schedule is legal. There are many legal schedules for the same program. For example, the logical execution time stamps for instances of $S1$ in Figure 2.16 can be

\[ \theta(i, j) = \frac{i(i - 1)}{2} + j \] (2.12)

Where $\theta$ represents the schedule function. However,

\[ \theta(i, j) = i \ast N + j \] (2.13)

is a legal schedule too, where $N$ is the upper bound for the loop index $j$. Usually, finding the affine schedule for a statement in a nested loop is challenging [31]. However, finding a multi-dimensional schedule for a statement is much easier. The very basic solution is to obtain the schedule from the abstract syntax tree (AST) directly [50, 31]. In the AST, all the loops are represented as inner nodes and all the statements are represented as leaf
01: for(int i=1; i <= N; i++)
02:   S1;
03: for(j=1; j <= M; j++)
04:   S2;
05: for(k=1; k < i+j; k++)
06:   S3;
07: S4;
08: for(int m=1; m <= i; m++)
09:   S5;
10: S6;

Figure 2.18: An example code segment with a loop

Figure 2.19: The AST for the code in Figure 2.18
nodes. The program itself can be treated as a very special loop that only iterates once.

The AST for the piece of code shown in Figure 2.18 is shown in Figure 2.19. The multi-dimensional schedule can be represented as an affine vector. For example, the schedule for S3 can be written as

\[
\theta(i, j, k) = (0, i, 1, j, 1, k, 0)
\]  

(2.14)

The multiple dimensional schedule for S3 can be obtained by recording the schedule values along the path from the root node to the statement node in the AST. The statements inside the same loop are assigned a constant value according to its order in the program [31].

For each statement we can represent all of its information by a polyhedron model with the iteration domain, memory access functions and a schedule. The polyhedron model is a solid mathematical model, and it is suitable to process and analyze the programs systematically. Finally, the programs can be recovered from the polyhedron representations easily.

2.2.4 Data Dependency Analysis Based On The Polyhedron Model

**Definition 12. Data dependency** [50, 29] A data dependency exists between two program statements (or instructions) S1 and S2 if and only if (1) both statements access the same memory location, and at least one of the data accesses is a write access, and (2) there is feasible run-time execution path from S1 and S2.

According to the Definition 12, we can see a data dependency has three elements:

1. Two statement instances: they can be the instances of different statements or the instances of the same statement within different loop iterations.

2. Both statement instances access the same location in the memory, which means both
of them have memory access expressions and each subscript of the memory array access are equal. Each of the memory accesses can be read or write access. We usually do not consider the input dependencies so at least one of the data accesses are a write access.

3. One of the instances executed precedes the other statement. We usually call the preceding statement as the data dependency source (represented by letter S) of the dependency and the latter one as the data dependency target (represented by letter T) of the dependency.

We can define a data dependency based on the polyhedron model [29] as

$$K_{S,T} = < (S, \bar{x}), (T, \bar{y}) > \quad \text{where} \quad \bar{x} \in D_S, \bar{y} \in D_T, (S, \bar{x}) \prec (T, \bar{y}), \bar{m}_S = \bar{m}_T \quad (2.15)$$

Where $\bar{x}$ and $\bar{y}$ are the iteration vectors of statement $S$ and $T$ respectively, $D_S$ and $D_T$ are the iteration domain of statement $S$ and $T$ respectively, $m_S$ and $m_T$ are the memory accesses in statement $S$ and $T$ respectively. The constraints can be combined together into a single polyhedron

$$< \bar{x}, \bar{y} > \in R_{S,T} \quad (2.16)$$

Where $R_{S,T}$ is the combined polyhedron of $D_S$, $D_T$ and the extra constrains $(S, \bar{x}) \prec (T, \bar{y})$ and $\bar{m}_S = \bar{m}_T$. For simplicity, we usually convert the data dependency representation shown in Definition 12 to the form of transform functions

$$\bar{x} = h(\bar{y}) \quad \text{where} \quad \bar{y} \in H_y \quad (2.17)$$

Where $h(\bar{y})$ is called a transform function [29], which can be used to get the iteration vector of the data dependency source from the iteration vector of the data dependency sink. $H_y$ is the iteration domain for the data dependency sink. Based on polyhedron model, the
Figure 2.20: An example code segment for data dependency analysis

transform function can be calculated by solving the linear programming problem [29, 30, 31, 28]:

$$
\max_{\vec{x}} \quad \vec{x} \\
\text{s.t.} \quad \vec{x} \in R_{S,T}
$$

(2.18)

We use an example code segment [29] in Figure 2.20 to show the work flow of the data dependence analysis. In the example code segment shown in Figure 2.20, we can see that the statement $S_1$ has memory accesses.

With the iteration vectors $\vec{x} = (i, j)$ and $\vec{y} = (i', j')$, the number of common loops is $N_{s_1,s_1} = 2$. $S_1$ is not before $S_1$, so $T_{s_1,s_1} = false$. For loop level $p = 0$, we have

$$Q_0 = \begin{cases}
\text{%iteration domains :} \\
0 \leq i \leq N \\
0 \leq j \leq N \\
0 \leq i' \leq N \\
0 \leq j' \leq N \\
\text{%memory access constraints :} \\
i + j = i' + j' \\
\text{%precedence constraints :} \\
i < i'
\end{cases}
$$

(2.19)
For loop level \( p = 1 \) we have

\[
Q_1 = \begin{cases}
\text{%iterationdomains :} \\
0 \leq i \leq N \\
0 \leq j \leq N \\
0 \leq i' \leq N \\
0 \leq j' \leq N \\
\text{%memoryaccessconstraints :} \\
i + j = i' + j' \\
\text{%precedenceconstraints :} \\
i = i' \\
j < j'
\end{cases}
\] (2.20)

For loop level \( p = 2 \), because \( T_{s_{1},s_{1}} = false \), \( Q_2 = \emptyset \). We can see that \( Q_1 = \emptyset \) too, because \( i = i' \) derives that \( j = j' \) using the memory access constraints, which violates \( j < j' \). According to (2.16), the data dependence can be constrained by the polyhedron \( Q_0 \). The corresponding transformation function can be calculated by solving the linear programming problem [29]:

\[
\max_{\bar{x}} \bar{x} \quad s.t. \quad \bar{x} \in Q_0
\] (2.21)

The solution of problem (2.21) is:

\[
\bar{x} = h(\bar{y}) = (i' - 1, j' + 1) \quad \text{where} \quad 1 \leq i' \leq N, 0 \leq j' \leq N - 1
\] (2.22)

As illustrated in Figure 2.21, when the dependence sink is \((i', j') = (2, 1)\) (The i-
Figure 2.21: Data dependency analysis (For N=5) [29]

operation vector marked by the blue circle), we have \( i' + j' = 3 \). So the dependence source \( \vec{x} = (i, j) \) must be located on the red dashed line in Figure 2.21. According to the precedence constraints, the feasible dependence sources are marked by the green box in Figure 2.21. We have to choose the last iteration as the dependence source, which is \( (i, j) = (1, 2) \), because all the previous memory writes to the same location are overridden by the latest one.
Chapter 3: Polyhedron Model For GPU Programs

3.1 Overview

The polyhedron model has been widely used in the modern compiler systems [29, 30, 31]. Many efforts have been made to develop the compiler frameworks [10, 57, 6] for automatic parallelization and improved data locality. In this chapter we will extend the original polyhedron model to perform data reuse analysis for GPU kernels. The working flow of our reuse analysis framework is illustrated in Figure 3.1.

As illustrated in Figure 3.1, the optimization compiler framework has a preprocessor and a post-processor. The preprocessor and the post-processor are based on the enhanced CETUS compiler framework proposed by Yang et al. [61, 36]. The preprocessor is used to preprocess the input GPU kernels to generate the intermediate code and gather information needed for data reuse analysis based on the polyhedron model. The detailed discussion of the preprocessor will be presented in Section 3.2. The polyhedron model is generated by a modified polyhedron compiler front end “clan” [8], and the polyhedron model for GPU kernels will be discussed in Section 3.3. Based on the polyhedron model we can do data reuse analysis. The optimization (such as loop tiling) can be applied based on the result of data reuse analysis. The polyhedron model will be transferred to the intermediate code by
Figure 3.1: The general work flow of our compiler framework

Before the data reuse analysis is applied, our preprocessor of the input programs will pre-process the input GPU kernels, so the input GPU kernels are ready for polyhedron model based analysis.

3.2 Preprocessor

Before the data reuse analysis is applied, our preprocessor of the input programs will pre-process the input GPU kernels, so the input GPU kernels are ready for polyhedron model based analysis.

First, in GPU systems, different types of memory accesses have different cache behaviors [2, 21]:

1. Registers, not cached.
2. Shared memory, not cached.

3. Local memory, cached by the L1 data cache: the write policies are write back on the cache hits and write-no-allocate on the cache misses.

4. Constant memory, cached by the read-only constant cache.

5. Texture memory, cached by the read-only texture cache.

6. Global memory, cached by L1 data cache: the write policies are write-evict and write-no-allocate.

Our preprocessor marks the memory type on all the memory access in the GPU programs to guide further memory access optimization. For example, we apply L1 cache optimization for the global memory accesses in our compiler framework. But shared memory accesses cannot be optimized by L1 cache optimization technique.

Second, in the GPU kernels, there are special variables representing the thread indexes (Denoted as 'tidx','tidy' for local thread indexes and 'idx','idy' for global thread indexes) and block indexes (Denoted as 'bidx' and 'bidy'). Like loop iteration variables, these special variables are also used as parameters that define the execution domain of a GPU kernel statement. However, these variables need to be distinguished from the ordinary loop iterators because of the parallel execution of the GPU kernels. The same statement with different thread id or block id can be executed in any order unlike the statements in different loop iterations. The statements in different loop iterations must be executed in lexicographic order [29]. Our preprocessor marks all those special variables on the IR of the GPU kernel codes. These marks will be used to construct the precedence constraints in the data reuse analysis stage.

The third preprocessing task is to parse the system parameters that will be used in our data reuse analysis. There are two types of parameters. (1) Hardware parameters: including the warp size, the L1 data cache size, the maximum number of CTAs etc.. These
parameters can be obtained from the database of the current GPGPU devices or specified by the users. (2) The GPU kernel parameters: including the block size and the grid size that can be parsed from the source code (require the users declare these sizes by pre-defined micro names such as BLOCK_SIZE_X) or specified by the users.

Finally, the original GPU kernels will be converted to the intermediate code as illustrated in Figure 3.2. Since the compiler front end can only generate the polyhedron models for the nested loops, so the normal GPU kernels must be converted to the intermediate code format in which parallel threads and thread blocks are represented as special loops. These loops are distinguished from the normal loops by special marks attached to the iterators by the second phase of the preprocessor we just discussed.
In Figure 3.2 ‘idx’, which is equal to ‘threadIdx.x + blockIdx.x * blockDim.x’, represents the global thread index along the x direction. ‘idy’, which is equal to ‘threadIdx.y + blockIdx.y * blockDim.y’, represents the global thread index along the y direction. Local thread indexes inside thread blocks are represented as ‘tidx’ along the x direction and ‘tidy’ along the y direction, which are short for ‘threadIdx.x’ and ‘threadIdx.y’. ‘gdimx’ and ‘gdimy’ represent the grid dimensions along the x direction and the y direction respectively. ‘bdimx’ and ‘bdimy’ are the block dimensions along the x direction and the y direction respectively. In the intermediate code, all the variables including the array variables will be assigned a decoration superscript. We have special marks for each variable in the GPU kernel code. A variable with the superscript $p$ means it is an iterator for special parallel loops, such as the thread block index and thread index. A variable with the superscript $r$ means it is a register variable. The superscript $g$ of a variable is used to represent the global variables. The superscript $s$ is used to represent the variables or arrays declared in the shared memory. The annotations at line 01 and line 15 in Figure 3.2(b) mark the scope of the polyhedron model based parsing.

With all these preprocessings, we can obtain the polyhedron representation of a GPU kernel based on the intermediate code. In our compiler framework, we choose the polyhedron front end ”clan” [8] written by C’dric Bastoal et al. as the parser to generate the polyhedron representation for a GPU kernel. We modify the polyhedron front end ”clan” to make it compatible with the special marks of the variables added by our preprocessor.

### 3.3 The Polyhedron Model for GPU Kernels

In this section, we extend the traditional polyhedron model to work with the GPU kernels. The polyhedral model of the GPU kernels is represented in OSL format [7]. Generally speaking, the execution vector of an instance of a statement $S$ nested in an $n$-level loop can be represented as
Figure 3.3: Micro benchmark

\[ \vec{e}_S = \begin{pmatrix} bidy & bidx & tidy & tidx & i_0 & i_1 & \ldots & i_{n-1} \end{pmatrix}^T \] (3.1)

where \( bidy \) and \( bidx \) denote the block indexes, \( tidy \) and \( tidx \) denote the thread indexes. \( i_0, i_1, \ldots, i_{n-1} \) are the loop iterators of the loops that nest around \( S \).

Figure 3.3 shows the GPU kernel source code for the matrix vector multiplication benchmark in the CUDA SDK. The global memory read accesses in \( S05 \) and \( S06 \) are cached. The execution instance of a statement can be represented with an execution vector including the thread block id, the thread id and the loop iterator of the statement.

A memory access nested in an \( n \)-level loop can be represented by the polyhedron constrained by the following inequalities (also defined as the execution domain):

\[
\begin{align*}
0 & \leq bidy < gdimy \\
0 & \leq bidx < gdimx \\
0 & \leq tidy < bdimy \\
0 & \leq tidx < bdimx \\
0 & \leq i_0 < end_0 \\
0 & \leq i_1 < end_1 \\
\ldots
\end{align*}
\] (3.2)
Where \( gdim_y \) and \( gdim_x \) are the grid sizes along the \( x \) direction and the \( y \) direction respectively, and \( bdim_y \) and \( bdim_x \) are the thread block sizes along the \( x \) direction and the \( y \) direction respectively. These constant values can be obtained by our preprocessor of the GPU kernels. \(\text{end}_0, \text{end}_1 \cdots\) are the loop boundaries which could be constants.

For example, the execution vector of \( S_{05} \) (\( a=A[\text{idx}][i] \)) can be represented as

\[
\overline{e}_{S_{05}} = \begin{pmatrix} bidy & bidx & tidy & tidx & i \end{pmatrix}^T
\]

The execution domain of the instances of statement \( S_{05} \) is

\[
\begin{cases}
0 \leq bidy < gdimy \\
0 \leq bidx < gdimx \\
0 \leq tidy < bdimy \\
0 \leq tidx < bdimx \\
0 \leq i < width \\
0 \leq width
\end{cases}
\]

It is easy to use the execution domain in (3.2) to represent the specified thread units. For example, we can add extra constraints involving \( tidx \) and \( tidy \) to represent the instances of a statement in a warp. As the warp size is 32 for Nvidia GPUs [22, 9], the warps belonging to warp \( w \) can be represented by the extra constraints (3.2):

\[
\begin{cases}
32w \leq bdimx \cdot tidy + tidx < 32(w + 1) \\
0 \leq w < \text{ceiling}(bdimx \cdot bdimy/32)
\end{cases}
\]

Where \( w \) is the warp index with the constraints \( 0 \leq w \leq \text{ceiling}(bdimx \cdot bdimy/32) \).

In addition to the execution domain, we need to know the access constraints of the
memory accesses before we apply data reuse analysis. An $m$-dimensional array access can be represented as a memory access vector [7]:

$$\vec{a}_S = (\alpha_0 \quad \alpha_1 \quad \ldots \quad \alpha_{m-1})^T$$  \hfill (3.6)

Where $\alpha_0, \alpha_1, \ldots, \alpha_{m-1}$ are the linear combinations of $\vec{e}_S$.

$$\alpha_k = c_0 * bidy + c_1 * bidx + c_2 * tidy + c_3 * tidx + c_4 * i_0 + \cdots$$  \hfill (3.7)

Where $c_0, c_1, \ldots$ are the constant coefficients. For example, the memory accesses of $S05$ (a=A[idx][i]) in Figure 3.3 can be represented as

$$\begin{aligned}
\alpha_0 &= bdimx * bidx + tidx \\
\alpha_1 &= i
\end{aligned}$$  \hfill (3.8)

When recovering programs form the polyhedron model, if we do not specify the schedule of each statement, there will be many ways to order the statements. In order to recover the GPU kernel form its polyhedron model representation with a polyhedral scanner like ClooG [13], we must obtain the schedule information [7] of the statements in the GPU kernels. According to the discussion in Section 2.2.3, the multiple level schedule information obtained based on the AST is a good choice to represent the execution order. In the GPU systems, the execution order of the parallel threads is not fixed. So the schedule cannot involve $bidx$, $bidy$, $tidx$ or $tidy$. However, the execution order of each thread is predictable just like the schedule of the normal C code. For example, the ordering information of the statement $S05$ in Figure 3.3 can be represented as

$$< 0, i, 0 >$$  \hfill (3.9)
Or in the matrix format

\[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & -1 \\
0 & 0 & 1 & 0
\end{pmatrix}
\begin{pmatrix}
t_1 \\
t_2 \\
t_3 \\
i
\end{pmatrix}
+ \begin{pmatrix}
0 \\
0 \\
0 \\
i
\end{pmatrix}
= \vec{0}
\]  \hspace{1cm} (3.10)

In (3.10), \(t_1, t_2,\) and \(t_3\) indicate the multi-dimensional iteration scheduling [31] of the statement \(S05\) in Figure 3.3.

### 3.4 Summary

In this chapter, we present the extended polyhedral model for GPU kernels. We can represent the relationship among parallel threads using the extended polyhedral model in a mathematical way, which made it a very powerful tool that can help us analyze the data reuse features during the execution of a GPU kernel.

Based on the polyhedral model for GPU kernels, we design an optimization framework consisting of four components to improve the performance of a GPU kernel automatically in the following chapters.
Chapter 4: Compiler-assisted

Programmable Warp Scheduler

4.1 Overview

During the execution of GPU programs, expensive operations such as off-chip DRAM accesses can be hidden by switching among a batch of threads that are ready to execute [12, 25, 42, 14]. The scheduling is performed on thread units called warps (also known as wavefronts). GPUs schedule a warp at a time, and a warp scheduler is designed to select which warp to execute when computation resources are available to take advantage of the computation ability of GPU processors and hide the memory latency.

In order to bridge the performance gap between computation, and off-chip DRAM memory accesses, each SM of the modern GPGPUs has on-chip L1 data cache [51, 32, 33, 47]. The L1 data cache improves the system performance only when data is reused during the execution process of the warps. There are two types of data localities during the execution process of GPU kernels.

The locality within the same warp is called intra-warp locality. The data locality among adjacent warps, is called inter-warp locality. Intra-warp data reuses occur when a memory access reuses the cache block brought in by another memory access issued by the same warp. Correspondingly inter-warp reuses are data reuses among different warps.
In GPU systems, the number of threads competing for the L1 cache at the same time is very large. If the cache blocks are evicted before they can be reused, then the overall performance will be degraded. Evictions will take place before the cache block can be reused by upcoming memory accesses (L1 cache thrashing). The default warp scheduler which issues warps in the round-robin manner is oblivious to intra-warp or inter-warp data locality, which would cause L1 cache thrashing [51, 32].

Many optimized warp schedulers have been proposed to improve the performance of the input GPU programs by preserving intra-warp and inter-warp data reuses [51, 33, 47]. Many researchers concluded that limiting the total number of warps issued simultaneously can reduce the total number of warps competing for the cache resources, and thus improve the overall system performance [51, 33, 47]. The basic ideas of those research works are to divide the warps into warp groups with different priorities, i.e., high priority warp group, and low priority warp group. When computation resources are available, the warp scheduler issues high priority warps first. The size of the high priority warp group affects the overall system performance significantly. When the warp group size is too small, there will not be enough warps to hide the long latency memory access operations. However, When the warp group is too large, the problem of L1 cache thrashing cannot be solved. The best warp group size can be determined by software profiling or by hardware, which is a challenging task. Profiling requires GPU programmers to tune their applications with different group sizes and choose the best group size manually. The tuning process may be time consuming and the profiling process must be repeated for different application programs. The other problem with profiling is different segments of a GPU program have different locality features. Some segments of a GPU program may have strong data reuses, so a smaller warp group size is preferred. Other segments of a GPU program will have better performance with a larger warp group size. However, hardware optimized scheduling techniques such as Cache-Conscious Wavefront Scheduling (CCWS) [51] can adjust the group size dynamically according to the memory access footprint with extra hardware support, which will
increase the chip area and overall power consumption.

In this chapter, we present a compiler and hardware assisted programmable warp scheduler to take advantage of the intra-warp locality and inter-warp locality simultaneously. The intra-warp and inter-warp data reuse analysis is based on the modified polyhedron model for GPU kernels [29, 30, 31, 10, 57, 6] as illustrated in Chapter 3. The programmable warp scheduler will decide when to limit the total number of warps issued, and the size of the high priority warp group. A source to source compiler pass is designed to perform intra-warp and inter-warp data reuse analysis, decide the best warp group size, and insert specific scheduler control instructions at proper positions automatically. Two extra PTX instructions, “setPriority” and “clearPriority”, are designed to support the programmable warp scheduler. The hardware overhead of our solution is very small: just some logic circuits to decode our new PTX instructions, a few extra registers to keep the status of the warp scheduler and tiny modifications of the original warp scheduler to filter out the high priority warps.

The rest of this chapter is organized as follows. Section 4.2 presents our design of the warp scheduler with high priority warp group. Section 4.3 presents the implementation of the programmable warp scheduler. Section 4.4 presents the compiler that supports the programmable warp scheduler. Section 4.5 reports the experimental results. Section 4.6 presents the related work. Section 4.7 concludes this chapter.

4.2 Warp Scheduler With High Priority Warp Groups

4.2.1 Problem Statement

For round-robin warp scheduler, the key factor for the performance loss is shown in Figure 4.1. Assume we have a direct mapped L1 data cache, whose capacity is two cache blocks. Each cache block stores 4 array items. Each warp has two threads. The original
Figure 4.1: The memory access pattern for matrix multiplication with the round-robin warp scheduler, in which $i$ represents the loop iteration

A round-robin scheduling algorithm will result in the memory block access order as shown in Figure 4.1, in which the numbers in the rectangular boxes indicate the order of the thread blocks being introduced. Assuming warp0 is first selected, block0 and block1 will be brought into cache. Then warp0 is stalled and warp1 is selected, so block2 and block3 will replace block0 and block1. At this point all the threads will be blocked. When warp0 is finally ready again, it still needs to access block0 and block1, but they have been replaced, so we need to bring in these two blocks into cache again, which is called thrashing in the L1 data cache. The thrashing in the L1 data cache will lead to poor performance.

In Figure 4.1, we can see that along each thread’s execution, we have good data locality, but the parallel execution breaks the locality. In other words, the parallel thread execution cannot take advantage of the data locality in the application program. That’s the main reason why the application program has poor performance.
Figure 4.2: The memory access trace with the round-robin scheduling algorithm. (The figure is zoomed in to show the details of a small portion of all the memory accesses occurred.)

Figure 4.2 shows the memory access pattern of the array \textit{data} collected from the benchmark matrix multiplication running on the GPGPU-sim. The size of the matrix is $1024 \times 1024$. The total number of the threads is 1024, and the number of the warps is 32.

In Figure 4.2, the x-axis represents the time when memory access instructions are issued and the y-axis represents the memory addresses accessed. We can see that large scale of memory locations are accessed at relatively similar time during each iteration, which leads to unnecessary L1 cache evictions. The thrashing in the L1 data cache cannot be avoided even though we have a relatively large L1 data cache.

4.2.2 Scheduling Algorithm

To solve the problem we have discussed in Section 4.2.1, we design the warp scheduler with high priority warp groups. The architecture of SMs with our warp scheduling algorithm is illustrated in Figure 4.3. The main idea of our warp scheduling algorithm is to keep the highest priority of the selected group for a longer time, then assign the highest priority to another warp group immediately after the warps in the highest priority warp group all
Figure 4.3: The architecture of an SM with the programmable warp scheduler. (The red items in ready queue and waiting queue indicate the warps with high priority. The modules with red boxes indicate the modules we have modified.)
block. By doing this, the same selected group can resume execution immediately after the warps in this group finish the long latency operations. This mechanism can effectively prevent other groups from seizing the highest priority too soon, which means a better use of the locality within the same warp.

Another improvement is when many warps are blocked in the waiting queue, the warps with the highest priority will be moved to the head of queue. Thus it guarantees the high priority warps are served first. By doing so, we can prevent unnecessary cache replacement for the warps in the waiting queue.

Our optimized scheduling algorithm includes two main parts: (1) An improved two-level scheduler for the ready queue. (2) A two-level scheduler for the waiting queue. The detailed scheduling algorithm is described as follows:

1) **Scheduler for the waiting queue** We design two waiting queues. One is for warps with high priority, and the other waiting queue is for warps with lower priority. Both waiting queues use a FIFO scheduling strategy. If the waiting queue with the high priority is not empty, then the scheduler always select warps from the high priority warp queue.

2) **Scheduler for the ready queue** The scheduling algorithm for the ready queue is illustrated as follows:

1. Divide all the warps into warp groups, and each group has the same size.

2. In the beginning, all the warps are assigned the same low priority.

3. The scheduler selects one warp using a round-robin algorithm, and sets all the warps in the same group to high priority. Also, all the warps in this group will keep their high priority until all the warps in this group are finished.

4. If all the warps in the high priority group stall, use the round-robin algorithm to select one of the warps with low priority and set it to high priority. The other warps will remain at a low priority.
5. If all the warps in the high priority group are finished, repeat step 3.

6. When a warp stalls, if it has high priority, put it into the high priority waiting queue; otherwise drop it into the low priority waiting queue.

The memory block access order of matrix multiplication algorithm using our warp scheduler is shown in Figure 4.4. In Figure 4.4, we assume warp 0 and 1 are in group 1, and warp 2 and 3 are in group 2. Assume group 1 will get the higher priority first, and it will keep this high priority until all the threads in this group are finished. Assume our cache has 2 cache blocks. For the access order of memory blocks shown in Figure 4.4, accesses of memory blocks 2–7 will have no cache misses. It will be the same situation for accesses of memory blocks 10–15. Using this warp scheduling algorithm, we can avoid most of the cache misses, which means improved performance.

We use a simple runtime scenario as shown in Figure 4.5 to illustrate how our scheduling algorithm works. We assume that each warp will access 4 memory blocks. If all the
Figure 4.5: Execution process for (a) the original scheduling algorithm. (b) Our scheduling algorithm. Assume the scheduling algorithm has 4 warps numbered from 0 to 3 for illustration purpose. The solid lines represent the execution of non-memory access instructions. The small circles represent memory access instructions. The dashed lines represent memory access instructions being served currently for this warp. Blank represents an idle instruction in this thread.
memory accesses are missed in the cache, this memory operation will take 4 time units. One more memory block hit in the cache means one less time unit will be used. We assume that warp 0 and warp 1 are in group 1, and warp 2 and warp 3 are in group 2. The size of the cache is assumed to be 4 cache blocks for simplicity. Memory accesses after another memory access in the same warp group will have 3 cache hits, and memory accesses after another memory access operation in different warp group will have no cache hit.

Figure 4.5(a) illustrates the execution process of the original scheduling algorithm. The scheduling algorithm first selects warp 0 then selects warp 2, which leads to 4 cache misses because these two warps are not in the same group, so the cache content will all be replaced. When finally warp 0 is selected to be executed again, the cache content is already replaced, which means it needs to bring in those memory blocks again. The total number of time units needed is 33 clocks.

But in Figure 4.5(b), our scheduling algorithm will first select the warps in group 1, which can reduce the time needed for each memory access to 1 time unit after the first memory access is finished. At the same time, when all the warps in group 1 are blocked, the scheduling algorithm can select the warps in other groups to execute while processing memory accesses for warp 1, and thus not waste the computation resources. And finally we can see the total number of time units needed is 15. Our warp scheduling algorithm saved nearly half of the execution time. If we have more warps to be scheduled, our warp scheduling algorithm can save even more time, since our scheduling technique can make the memory access pattern more regular as shown in Figure 4.6.

In Figure 4.6, we can see that our warp scheduling algorithm makes the memory access more regular. Each warp group’s memory accesses are performed in a relatively concentrated time scale. And the data reuses are enhanced.

In our warp scheduling algorithm there are only two priority levels for each warp. So we can use one bit to represent the priority level for each warp, shown as the shaded part in the queue in Figure 4.7. Then the priority queue can be implemented as shown in
Figure 4.6: The memory access pattern trace with our scheduling algorithm.

Figure 4.7: Hardware architecture of the priority queue.

The priority queue can be easily implemented by using a modified encoder. The input of the encoder is all of the priority bits in the queue. If there are warps in the queue with high priority, this encoder will give out the position of the first warp in the queue with high priority, which will then be selected. If there is no warp in the queue with high priority, it will give out a none signal. The first warp in the queue will be selected. The encoder is a very simple digital circuit module which does not consume much extra chip area.
4.3 Programmable Warp Scheduler

With the warp scheduler based on high priority queue, we can control the behavior of the warp scheduler to preserve intra-warp and inter-warp locality. However, there is still another problem, i.e., how to control the high priority warp group size. In this section, we present a programmable warp scheduler to let the software control the behavior of the warp scheduler through our new designed warp scheduler control instructions.

4.3.1 Warp Priority Register and Warp Priority Lock Register

Two extra special registers are added for the programmable warp scheduler: the warp priority register and the warp priority lock register. The warp priority register is a bit register in which each bit represents the priority of the corresponding ready warp. If a warp is issued with high priority, the corresponding bit is set to 1. When the warp scheduler issues new warps, it will first check the priority register and the warps with high priority will be issued. The width of the warp priority register is determined by the maximum number of warps that can be executed simultaneously, which is relatively small. The content of the warp priority register can be modified by two special PTX instructions “setPriority” and “clearPriority” designed for our programmable warp scheduler.

According to the SIMD execution module of the GPU SMs, programs executed in each warp are identical to each other. So every warp has an equal chance to modify the content of the warp priority register. A lock must be used to prevent other warps from setting the warp priority register before the warps in the high priority group have finished. Our lock for the programmable warp scheduler is a register with two states, locked and free. When the lock is free, any warp can lock it by issuing “setPriority” instruction. The warp ID will be recorded at the same time. When the lock is locked, all the other warps cannot lock it, change the lock bit, or update the content in the warp priority register. The lock can only be unlocked by the “clearPriority” instruction issued by the same warp or
4.3.2 Design of the Instructions “setPriority” and “clearPriority”

Because CTAs are dynamically assigned at runtime, the compiler cannot know the absolute warp id of the current warp and the warps that gain high priority. The compiler can only compute the size of the high priority warp group. Then, the hardware will utilize this information to update the warp priority register according to the group size and the warp ID that issues the “setPriority” instruction. Assume that the group size is $N > 1$, the current warp and the $N-1$ warps with consecutive $N-1$ warp ids following the current warp id will be in the same warp group as the current warp, and the warp group including the current warp will be given high priority. For example, assume 8 active warps 0 to 7 on an SM are active. The hardware warp IDs are illustrated in Figure 4.8. Assume the “setPriority” instruction is issued by warp 0 and the group size is 4, then warps 0, 1, 2, 3 will gain high priority.

The execution processes with and without high priority group set are illustrated in Figure 4.9. Assume we have 4 warps ready for execution on an SM. In each warp, several memory accesses are executed during the shadow part of the execution process as shown in Figure 4.9. When 4 of them are issued with equivalent priority, the competition for L1 cache resources leads to many unnecessary cache block evictions. The overall execution
Figure 4.9: An warp issue example. (a) Round robin warp scheduler, (b) Programmable warp scheduler
time is relatively long, as shown in Figure 4.9(a).

With the programmable warp scheduler, we can reduce the chance of L1 cache thrashing. Assume the cache capacity can hold two warps for illustration purpose, the programmable warp scheduler will set the high priority warp group size to two based on the data reuse analysis. Based on the analysis result, the compiler can insert setPriority, before memory accesses start, to construct a high priority group with two warps and destroy the high priority group when the memory accesses are finished. The execution process with programmable scheduler is shown in Figure 4.9(b). In Figure 4.9(b), the high priority group are first constructed by warp 0, and only warp 0 and warp 1 can be issued after this point. Because of less competitions for the L1 resource, the total number of cache block evictions in L1 cache is reduced and the overall performance is improved. The performance is also improved for warp 2 and warp 3. Protected by the warp priority lock, high priority warp group will not be reset during the execution of each high priority group.

4.4 Compiler Supporting The Programmable Warp Scheduler

With the programmable warp scheduler we can control the behavior of warp scheduler during the execution of GPU kernels. However, there are still two unresolved problems. (1) When should we set up and destroy the high priority warp group? (2) How large the high priority warp group should be to obtain the best performance? From Section 4.2, we can reach the conclusion that programmable warp schedule can only improve the performance of L1 data cache when intra-warp data reuses or inter-warp data reuses exist. So the answer to the first question is that we set up the high priority warp group only when there are intra-warp or inter-warp data reuses in L1 cache. For the second question, our experiment shows that the size of high priority warp group do affect the performance of the L1 data cache.
Figure 4.10 reports the simulation results of the 2D convolution benchmark with different high priority warp group size (warp limiting) and different L1 cache sizes to show the importance of the size of the high priority warp group in avoiding cache thrashing, and performance cliff. From Figure 4.10, we see that when the high priority warp group size is small, the overall performance increases when the high priority group size increases, since memory access latency can be better hidden when more warps can be scheduled. However, the overall performance drops significantly when the size of the high priority warp group is larger than a certain threshold, and this phenomenon is called “performance cliff” [53, 54], where a slight increase of the number of warps competing for cache resources can result in significant performance degradation. The “performance cliff” is caused by L1 cache thrashing. To solve the second problem, and avoid the “performance cliff”, we develop a compiler framework to estimate the best high priority warp group size automatically and a programmable warp scheduler via scheduler control instructions in this chapter.

When the L1 cache size is 32K, no “performance cliff” is observed, since the L1 cache is no longer the performance bottleneck. In this case, the high priority warp group is no longer needed. Our compiler framework can detect this situation and would not degrade the performance of warp scheduler in this situation.
4.4.1 Intra-Warp Data Reuse Detection

As illustrated in Figure 4.11, intra-warp data reuses occur when a memory access reuses the cache block brought in by another memory access issued by the same warp. Intra-warp data reuse can be formally defined as

**Definition 13.** Intra-warp data reuse During the execution process of a thread block, if there exists two memory accesses $M$ and $M'$ that meet all of the following conditions, then intra-warp data reuse exists in the thread block.

1. The thread that issues $M$ and the thread that issues $M'$ are within the same warp.

2. $M'$ is issued later than $M$.

3. $M'$ reuses the data in the L1 cache brought in by $M$.

Figure 4.11: Intra-warp data reuses (The red dots represent global memory accesses)
Condition 1 in Definition 13 can be represented by inequalities:

\[
\begin{align*}
0 & \leq b_{idy}, b_{idy}' < g_{dimy} \\
0 & \leq b_{idx}, b_{idx}' < g_{dimx} \\
0 & \leq t_{idy}, t_{idy}' < b_{dimy} \\
0 & \leq t_{idx}, t_{idx}' < b_{dimx} \\
32w & \leq b_{dimx} * t_{idy} + t_{idx} < 32(w + 1) \\
32w' & \leq b_{dimx} * t_{idy}' + t_{idx}' < 32(w' + 1) \\
0 & \leq w, w' < \text{ceiling}(b_{dimx} * b_{dimy}/32) \\
w & = w'
\end{align*}
\]  

(4.1)

Where \(b_{idy}, b_{idx}, t_{idy}, t_{idx}\) are thread indexes and thread block indexes of \(M, t_{idy}', b_{idx}', t_{idy}', t_{idx}'\) are thread indexes and thread block indexes of \(M'\). \(w\) and \(w'\) are warp indexes of \(M\) and \(M'\).

The second condition can be represented with the sequencing predicate [29]. We have

\[M \prec M'\]  

(4.2)

which is equivalent to

\[M \prec_p M'(p = 0, 1, 2....N)\]  

(4.3)

for each possible iteration depth \(p\). Assume there are \(N\) common loops nesting around both \(M\) and \(M'\) for illustration purpose. The iteration vector of those common loops is the sub-vector of the execution vector of \(M'\) and \(M\):

\[\vec{i}' = \left( i'_0, i'_1, \ldots, i'_{N-1} \right)^T \subseteq \vec{e}', \vec{i} = \left( i_0, i_1, \ldots, i_{N-1} \right)^T \subseteq \vec{e}\]  

(4.4)
Then the constraints to ensure the precedence order are illustrated as follows [29]:

\[
\begin{cases}
\vec{v}[0..p] = \vec{v}[0..p] \land \vec{v}[p + 1] > \vec{v}[p + 1] & \text{when } 0 \leq p < N \\
\vec{v}[1..N - 1] = \vec{v}[1..N - 1] \land T_{M',M} = true & \text{when } p = N
\end{cases}
\]  (4.5)

Where \( T_{M',M} \) is true when \( M \) textually precedes \( M' \). \( T_{M',M} \) is false when \( M' \) and \( M \) are issued by the same statement.

In order to meet the last condition in Definition 13, the addresses of memory accesses \( M' \) and \( M \) are mapped to the same L1 cache block with the same cache block tag. At compile-time, we cannot know the physical addresses of the memory accesses, but we can estimate the memory access pattern with the array access subscripts. Without loss of generality, we assume that arrays are aligned to cache blocks and data reuses in the L1 cache only exist along the last dimension of array accesses.

For example, assume cache block size is 128 Bytes, and each element in a two dimensional array \( A[64][64] \) is 4 Bytes long for illustration purpose. Then each cache block can hold 32 array elements. If \( M \) accesses \( A[7][11] \) and \( M' \) accesses \( A[7][15] \) after \( M \), then \( M' \) can reuse the data in the L1 cache brought in by \( M \) (If the cache block has not been evicted). However, if \( M' \) accesses \( A[7][32] \) or \( A[8][11] \), it is impossible for \( M' \) to reuse the data in the L1 cache brought in by \( M \) based on our assumption.

If we denote the number of array elements in a cache block of L1 cache as \( B \), then the constraints representing condition 3 for \( m \)-dimensional array accesses in Definition 13 are shown as follows:

\[
\begin{cases}
\vec{a}'_i = \vec{a}_i, i = 0...m - 2 \\
\vec{a}_{m-1}/B = \vec{a}_{m-1}/B
\end{cases}
\]  (4.6)

In (4.6) “/” indicates integer division. In order to represent (4.6) in the form of inequalities, we have to introduce four intermediate variables represented as a vector \( \vec{v} \):

\( \vec{v} = (\beta', \beta, \gamma', \gamma)^T \), where \( \beta' \) and \( \beta \) indicate the cache block indexes of \( M' \) and \( M \), and
\(\gamma\) and \(\gamma\) indicate the offsets inside each cache block. (4.6) can be rewritten as inequalities:

\[
\begin{aligned}
\vec{\alpha}'_i & = \vec{\alpha}_i, i = 1 \ldots m - 2 \\
\vec{\alpha}'_{m-1} & = \beta' * B + \gamma' \\
\vec{\alpha}_{m-1} & = \beta * B + \gamma \\
\beta' & = \beta \\
-B + 1 & \leq \gamma' - \gamma \leq B - 1 \\
0 & \leq \gamma' \leq B - 1 \\
0 & \leq \gamma \leq B - 1 \\
0 & \leq \beta' \\
0 & \leq \beta 
\end{aligned}
\] (4.7)

When intra-warp data reuse exists at iteration level \(p\), the intra-warp data reuse distance be defined as the lexicographic distance between \(\vec{i}'\) and \(\vec{i}\):

\[
\vec{\delta}_p = \vec{i}' - \vec{i}
\] (4.8)

The lexicographic maximum of \(\vec{\delta}_p\) is the longest life time that a cache block is held in L1 cache without eviction to preserve the intra-warp data reuse between \(M'\) and \(M\).

The problem of detecting intra-warp data reuses at iteration level \(p\) can be represented as the following integer linear programming problem:

\[
\begin{aligned}
\max & \quad \vec{\delta}_p \\
\text{s.t.} & \quad \vec{\delta}_p = \vec{i}' - \vec{i} \\
& \quad \vec{i}', \vec{i} \in Q_p 
\end{aligned}
\] (4.9)
$Q_p$ is the polyhedron defined by formula (4.1), (4.5), (4.7) and (3.7).

For any memory access pair, $M$ and $M'$, we can detect whether or not there are intra-warp data reuses between them with (4.9). If (4.9) has a solution, there are intra-warp data reuses between $M$ and $M'$ at iteration level $p$. Otherwise, there is no intra-warp data reuses among them. The problem (4.9) can be solved with the integer linear programming system PipLib [29, 28] developed by Feautrier et al.

### 4.4.2 Inter-Warp Data Reuse Detection

As illustrated in Figure 4.12, inter-warp reuses are data reuses among different warps, which can be formally defined as:

**Definition 14. Inter-warp data reuse** During the execution process of a thread block, if there exists two memory accesses, $M$ and $M'$, which meet all of the following conditions, then inter-warp data reuse exists in the thread block.

1. The thread that issues $M$ and the thread that issues $M'$ are in different warps.
2. $M'$ reuses data in the L1 cache brought in by $M$.
There are two differences between the detection of inter-warp data reuse and intra-warp data reuse:

1. The execution order of the warps in an SM is controlled by the hardware warp scheduler, so the lexicographic precedence order of $M$ and $M'$ in the source code can no longer determine the execution order of instances of $M$ and $M'$. So the precedence order constraints (4.5) required in intra-warp data reuse detection system are no longer needed.

2. The warp constraints (4.1) is modified to indicate that $M'$ and $M$ are in different warps:

\[
\begin{align*}
0 & \leq bidy, bidy' < gdimy \\
0 & \leq bidx, bidx' < gdimx \\
0 & \leq tidy, tidy' < bdimy \\
0 & \leq tidx, tidx' < bdimx \\
32w & \leq bdimx * tidy + tidx < 32(w + 1) \\
32w' & \leq bdimx * tidy' + tidx' < 32(w' + 1) \\
0 & \leq w, w' < \text{ceiling}(bdimx * bdimy/32) \\
0 & < w - w'
\end{align*}
\] (4.10)

If inter-warp data reuse exists, the target parameter that we would be interested in is the distance of inter-warp reuse, which is defined as:

\[
\xi = |w' - w| 
\] (4.11)

Where $w'$ and $w$ are the warp indexes of $M'$ and $M$. The problem of detecting inter-warp data reuse can be represented as the following integer linear programming problem:
\[
\begin{align*}
\max & \quad \xi \\
\text{s.t.} & \quad \xi = |w' - w| \\
& \quad w', w \in F
\end{align*}
\] (4.12)

Where \( F \) is the polyhedron defined by formula (4.10), (4.7) and (3.7).

If problem (4.12) has no solution, there is no inter-warp data reuse. Otherwise, there is inter-warp data reuses between \( M' \) and \( M \) and the maximum reuse distance is \( \xi_{\text{max}} \).

4.4.3 Group Size Upper Bound Detection

The “performance cliff” is caused by self-evictions in the L1 cache due to the parallel execution feature of GPU threads. A single memory access statement in source code can generate many memory access instances. We call those memory access instances concurrent memory accesses. Only the active warp generates memory access instances. So the number of memory access instances issued simultaneously is affected by the size of high priority warp group. Self-eviction occurs when the L1 data cache occupied by the concurrent accesses of the same statement is larger than the size of L1 data cache.

As illustrated in Figure 4.13, we formally define the concurrent memory accesses as

**Definition 15. Concurrent Memory Accesses** During the execution process of a thread block, two memory accesses \( M \) and \( M' \) are concurrent memory accesses if they meet all of the following conditions:

1. \( M \) and \( M' \) are issued by the same statement.

2. \( M \) and \( M' \) are issued by the warps located in the high priority warp group.

3. \( M \) and \( M' \) are issued simultaneously, which is represented as the iteration vectors for \( M \) and \( M' \) are equal to each other.
In Definition 15, the number of total memory requests issued simultaneously is determined by the high priority warp group size $\theta$. Intuitively, the larger the $\theta$, the more memory requests are issued simultaneously and more L1 cache blocks are occupied. When the number of cache blocks occupied by the memory accesses issued simultaneously is larger than the capacity of the L1 cache, evictions will take place before the cache block can be reused by upcoming memory accesses. We call this kind of cache eviction self-eviction. Self-evictions should be avoided because they increase the L1 cache misses and reduce the overall performance. So we define the minimum group size that will cause self-eviction as the upper bound of the warp group size. Our experimental results in Figure 4.14 show self-evictions are the main reasons for “performance cliff”. In Figure 4.14, we present the relationship between the performance and the high priority warp group size when the effect of self-evictions is eliminated. To eliminate the effect of the self-evictions we design a special victim cache that will record the base address of an array access and make all the memory accesses hit in the victim cache if they belong to the same array. Compared to Figure 4.10, we can see the “performance cliff” has been avoided.
Figure 4.14: Performance vs. group size without the effect of self-evictions (This simulation result measured by running the 2D convolution benchmark on GPGPU-sim with high priority warp group controlled by the programmable warp scheduler)

The problem can be solved if we can obtain the number of cache blocks occupied by a set of concurrent memory accesses with a specified warp group size $\theta$. The cache blocks occupied by an $m$-dimensional array can be estimated with

$$O_\theta = \max(|\beta' - \beta| + 1) \cdot \prod_{i=1}^{m-1} \max(|\alpha'_i - \alpha_i| + 1)$$  \hspace{1cm} (4.13)

Where $\beta'$ and $\beta$ are cache block indexes defined in (4.7) and $\alpha'_i$ and $\alpha_i$ are the elements in the memory access vector (3.6). Since we assume that the adjacent elements in the last dimension of an array can be reused in a cache block, so the number of cache blocks occupied along the last dimension of the array is defined by the largest distance among cache block indexes ($\beta$s). For the remaining dimensions, the number of cache blocks occupied along each dimension is the maximum distance between array indexes along that dimension.

According to Definition 15, some modifications are made in $Q_p$ to meet the requirements for concurrent memory accesses.

1. Cache block reuse constraints are no longer needed, and the accessed cache blocks
without data reuse constraints would be represented:

\[
\begin{cases}
\alpha'_i = \alpha_i, i = 1 \ldots n - 1 \\
\alpha'_n = \beta' * B + \gamma' \\
\alpha_n = \beta * B + \gamma \\
0 \leq \gamma' \leq B - 1 \\
0 \leq \gamma \leq B - 1 \\
0 \leq \beta'
\end{cases}
\] (4.14)

2. Iteration vectors of \(M'\) and \(M\) must be equal to each other:

\[
\vec{i}' = \vec{i}
\] (4.15)

Where \(\vec{i}'\) and \(\vec{i}\) are defined in (4.4).
3. Warp constraints are defined with the priority group size $\theta$:

\[
\begin{align*}
0 & \leq \text{bidy}, \text{bidy}^\prime < \text{gdimy} \\
0 & \leq \text{bidx}, \text{bidx}^\prime < \text{gdimx} \\
0 & \leq \text{tidy}, \text{tidy}^\prime < \text{bdimy} \\
0 & \leq \text{tidx}, \text{tidx}^\prime < \text{bdimx} \\
32w & \leq \text{bdimx} \times \text{tidy} + \text{tidx} < 32(w + 1) \\
32w^\prime & \leq \text{bdimx} \times \text{tidy}^\prime + \text{tidx}^\prime < 32(w^\prime + 1) \\
0 & \leq w, w^\prime < \text{ceiling}(\text{bdimx} \times \text{bdimy}/32) \\
0 & \leq |w - w^\prime| \leq \theta
\end{align*}
\]

(4.16)

Then $\delta_{m-1} = \max(|\beta^' - \beta| + 1)$ can be calculated by solving the optimization problem:

\[
\begin{align*}
\max & \quad \delta_{m-1} \\
\text{s.t.} \quad & \delta_{m-1} = \max(|\beta^' - \beta| + 1) \\
& \beta^', \beta, \alpha^i, \alpha_i \in E
\end{align*}
\]

(4.17)

Where $E$ is the polyhedron defined by equation (3.7, 4.14, 4.15, 4.16).

$\delta_i = \max(|\alpha^i - \alpha_i| + 1)$ can be calculated by solving the optimization problem:

\[
\begin{align*}
\max & \quad \delta_i \\
\text{s.t.} \quad & \delta_i = \max(|\alpha^i - \alpha_i| + 1) \\
& \alpha^i, \alpha_i \in T
\end{align*}
\]

(4.18)

Where $T$ is the polyhedron defined by equation (3.7, 4.15, 4.16).

With $O_\theta$ obtained, we can initialize $\theta$ to be 1, and try to increase $\theta$ by 1 each time, until $O_\theta$ is larger than the total number of cache blocks in the L1 cache. Then, we record
the final \( \theta \) value as the group size boundary, \( \theta_b \).

### 4.4.4 Putting It All Together

The compiler supporting the programmable warp scheduler must solve the following problems:

1. Whether or not the high priority warp group is needed for a code section in a GPU kernel.
2. When the high priority group is needed, how large the group size is.
3. When the high priority group is needed, where we insert “setPriority” and “clearPriority” instructions.

In order to simplify our implementation, we check whether or not the high priority warp group is needed for each loop. There are four major steps in Algorithms 1.

**Step1** Line 1–2, recursively process nested loops.

**Step2** Line 3–13, obtain the minimum high priority warp group size boundary and the minimum inter warp reuse distance.

**Step3** Line 14–17, put scheduler control instructions and set high priority warp group size. If the minimum high priority warp group size boundary and the minimum inter warp reuse distance are larger than the number of warps in thread block \( \#warpsInBlock \), high priority warp group is no longer needed since the warp scheduler is no longer the performance bottleneck.

**Step4** Line 18–19, clear scheduler control instructions on inner nested loops.
Algorithm 1: Insert warp scheduler control instructions

Input: A loop in GPU kernel L
Output: A loop with scheduler control instructions surrounded

for All the inner nested loops l do
  Apply this algorithm recursively on l;
end

minInterDistance = ∞;
minBoundarySize = ∞;

for All the memory access pairs M and M' in L do
  Calculate intra-warp reuse distance intraDistance (Equation 4.9);
  Calculate inter-warp reuse distance interDistance (Equation 4.12);
  if interDistance < minInterDistance then
    minInterDistance = interDistance;
  end
  if intraDistance > 0 or interDistance > 0 then
    Get high priority warp group size boundary boundarySize (Equation 4.13);
    if boundarySize < minBoundarySize then
      minBoundarySize = boundarySize;
    end
  end
end

if min(minInterDistance, minBoundarySize) < #warpsInBlock then
  Put ‘setPriority’ before loop L;
  Set group size to min(minInterDistance, minBoundarySize);
  Put ‘clearPriority’ after loop L;
  for All the inner nested loops l do
    Clean ‘setPriority’ and ‘clearPriority’ around l;
  end
end
__device__ inline unsigned int setPriority(unsigned int groupSize) {
    unsigned int ret;
    asm("bfind.u32 %0, %1; ":=r"(ret):'r"(groupSize));
    return ret;
}

Figure 4.15: Implementation of setPriority()

4.5 Experiments

4.5.1 Baseline Hardware Configuration and Test Benchmarks

We configure GPGPU-sim developed by Aamodt et al. [2, 5] to simulate the GTX480 GPU as the test platform to evaluate our programmable warp scheduling technique for GPGPUs. The detailed configuration for the baseline GPU system is shown in Table 4.1. We use NVCC 4.2 to compile the output source code of our source-to-source compiler framework. NVCC is not an open source compiler, so it is challenging for us to insert two new instructions “setPriority” and “clearPriority” in NVCC. So we borrow a rarely used existing instruction supported by NVCC to implement our “setPriority” and “clearPriority” instructions without modifying NVCC. We choose the “bfnd” instruction to implement our “setPriority” and “clearPriority” instructions. The “setPriority” and “clearPriority” instructions can be implemented by an inline function as illustrated in Figure 4.15. The argument of the function “setPriority” is the warp group size. When the argument is zero, the function “setPriority” would be interpreted as “clearPriority”. Necessary modifications are made in GPGPU-sim to enable it to decode our programmable warp scheduler control instructions, “setPriority” and “clearPriority”.

The benchmarks used in our experiments are obtained from the CUDA SDK and Rodinia parallel benchmark suite [15]. The descriptions and input sizes of the input benchmarks are summarized in Table 4.2.
<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
<th>Number of SMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of SMs</td>
<td>15 (as Nvidia GTX480)</td>
<td></td>
</tr>
<tr>
<td>Number of integer processing units per SM</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Number of floating point processing units per SM</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SIMD width</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>Size of L1 data cache</td>
<td>16KB, 4 way associative</td>
<td></td>
</tr>
<tr>
<td>Size of L2 data cache</td>
<td>512KB, 8 way associative</td>
<td></td>
</tr>
<tr>
<td>Size of shared memory</td>
<td>48KB</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: The baseline simulator configuration

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Input size</th>
<th>#kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td>mv</td>
<td>Matrix vector multiplication</td>
<td>4Kx4K</td>
<td>1</td>
</tr>
<tr>
<td>imergionmax</td>
<td>Image process algorithm</td>
<td>1Kx1K</td>
<td>1</td>
</tr>
<tr>
<td>conv</td>
<td>Two dimensional convolution</td>
<td>4Kx4K</td>
<td>1</td>
</tr>
<tr>
<td>kmeans</td>
<td>A cluster algorithm that is widely used in the field of data-mining</td>
<td>494020 vectors</td>
<td>2</td>
</tr>
<tr>
<td>hotspot</td>
<td>Scientific simulation program</td>
<td>1Kx1K</td>
<td>1</td>
</tr>
<tr>
<td>bfs</td>
<td>The breadth first Search algorithm of graph</td>
<td>1M</td>
<td>2</td>
</tr>
<tr>
<td>backprop</td>
<td>Back propagation algorithm used in the training process of layered neural networks</td>
<td>64K</td>
<td>2</td>
</tr>
<tr>
<td>nn</td>
<td>The K-Nearest neighbors algorithm which is widely used as classifier</td>
<td>10K</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.2: The benchmarks used in the evaluation [15]
4.5.2 Group Size Estimation Accuracy

Our compiler framework estimates the best group size for the high priority warp group with the information available at compile-time. We consider the most significant aspects that would affect the runtime performance, i.e., the global memory accesses and the L1 data cache size. We do not consider other factors such as interconnection networks, ALUs, and memory access units that would influence the best group size for the warp groups. In this subsection, we evaluate the estimation accuracy of the warp group size by our compiler framework.

The IPC (instructions per clock cycle) performance results, with different warp group sizes for conv benchmark are reported in Figure 4.10. The performance results depend on the L1 cache size. As shown in Figure 4.10, when L1 cache size is 8K, the group size estimated by our compiler is 8 and the group size with the best performance is 8. The estimation of the best group size by our compiler is correct. When L1 cache size is 16K, the group size estimated by our compiler is 16, but the group size with the best performance is 17. We define the accuracy of estimation as $acc = \frac{IPC_{estimated}}{IPC_{best}} \times 100\%$. For the 16K L1 cache case, the accuracy is $816.9/819.6 \times 100\% = 99.7\%$. When L1 cache size is 32K, the L1 cache is no longer the performance bottleneck. Our estimation of the group size, 32, still achieves an accuracy of 99.2%.

The group size upper bound detected in Subsection 4.4.3 can prevent the "performance cliff" from occurring as illustrated in Figure 4.10. The compiler estimation accuracy is summarized in Table 4.3. As shown in Table 4.3, our compiler framework can estimate the best warp group size with accuracy higher than 95%.
<table>
<thead>
<tr>
<th>Benchmark name</th>
<th>Estimation accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mv</td>
<td>100</td>
</tr>
<tr>
<td>imergionmax</td>
<td>99.7</td>
</tr>
<tr>
<td>conv</td>
<td>95.6</td>
</tr>
<tr>
<td>kmeans kernel 1</td>
<td>100</td>
</tr>
<tr>
<td>kmeans kernel 2</td>
<td>99.1</td>
</tr>
<tr>
<td>hotspot</td>
<td>95.2</td>
</tr>
<tr>
<td>bfs kernel 1</td>
<td>98.7</td>
</tr>
<tr>
<td>bfs kernel 2</td>
<td>98.2</td>
</tr>
<tr>
<td>backprop kernel 1</td>
<td>96.4</td>
</tr>
<tr>
<td>backprop kernel 2</td>
<td>95.8</td>
</tr>
<tr>
<td>nn</td>
<td>98.5</td>
</tr>
</tbody>
</table>

Table 4.3: The estimation accuracy

Figure 4.16: Speedups
4.5.3 Experimental Results

We compare the performance of our programmable warp scheduler (PWS) with the following four existing warp scheduling techniques:

(1) LRR (Loose round-robin): The default warp scheduler adopted by the GPU system. The scheduler always chooses the next warp in a circular order in the ready warp pool.

(2) Two-level: The warp scheduling technique proposed by Narasiman et al. [47], which divides the warps into multiple warp groups of fixed size. The scheduler chooses one group in the high priority warp group and always issues the warps in the high priority warp group until all of the warps in the high priority warp group are stalled.

(3) GTO: Greedy then oldest algorithm, the greedy version of the two-level warp scheduler proposed by Gebhars et al. [33]. The scheduler always chooses the warp recently issued. Only when it is stalled, the warp with the oldest time stamp will be chosen.

(4) CCWS: The cache-conscious wavefront (warp) scheduler proposed by Rogers et al. [51]. It limits the size of the high priority warp group by an automatic scoring system based on the victim cache.

As shown in Figure 4.16, our programmable warp scheduler improves the overall performance of the input benchmarks by $6\% - 230\%$ compared to the round robin scheduling technique. The kernels, imeroinmax, mv, and kmeans_kernel1, contain uncoalesced global memory accesses, so our programmable warp scheduler can improve the overall performance significantly by taking advantage of inter-warp data reuses (172.2\% for mv, 116.7\% for imeroinmax, and 178.3\% for kmeans_kernel1). The benchmark, conv, has a very low warp group size boundary, as illustrated in Figure 4.10. When the group size is larger than the boundary, the performance decreases significantly due to self-evictions. Our compiler can limit the high priority warp group size with the self-eviction boundary, which leads to significant performance gain (238.1\%). The remaining kernels are optimized with coalesced global memory accesses, so the performance improvements are not as significant as the uncoalesced kernels, yet the overall performance is still improved by $6\% - 70\%$. 
Compared to the two-level or GTO scheduling techniques, our programmable warp scheduler can achieve better performance by taking advantage of the long term high priority warp group. CCWS [51] maintains long term high priority warp group, but it will spend time training its scoring system. Without the training overhead, our programmable warp scheduler can improve the performance by $4\% \sim 11\%$ compared to CCWS.

The relationship between the L1 cache size and the performance for the test benchmark, kmeans_kernel1, is reported in Figure 4.17. We can see that as the L1 data cache size increases, the overall performance is improved by all the scheduling techniques. But the performance improvement by the round robin warp scheduler is only $4\%$ (when the L1 data cache size is increased from $8KB$ to $64KB$), because it cannot take advantage of the data reuses in the L1 data cache. The two-level and GTO scheduling techniques can achieve better performance improvement compared to the round robin warp scheduler. However, limited by the short term high priority warp group, they cannot fully take advantage of the data reuses in the L1 data cache. Our programmable warp scheduler, like CCWS, can fully take advantage of the L1 data cache by preserving intra-warp data reuses, and inter-warp data reuses. The performance is improved significantly when the L1 data cache size increases. When the cache size is larger than $32KB$, the performance can no longer be improved because L1 data cache is no longer the performance bottleneck.
4.6 Related Work

Many optimized warp scheduling techniques have been proposed to improve the performance of GPGPU architectures [51, 33, 47, 45]. Rogers et al. [51] proposed a long term warp throttle algorithm. Their scheduling techniques can preserve intra-warp data reuses by automatically determining the best high priority warp group size at runtime. However, inter-warp data reuses are not considered, because there are no relationships among the warps that can be issued. The other drawback of their scheduling techniques is that it needs extra hardware support, i.e., a victim cache is used to monitor the locality features of the applications, which has hardware overhead. In addition, the training process of their adaptive algorithm can also lead to performance loss. Compared to their work, our programmable warp scheduler enhances the intra-warp data reuses and inter-warp data reuses by applying data reuse analysis, and chooses the best group size for the high priority warp group with the compiler’s support. The hardware overhead of our programmable warp scheduler is tiny (Our programmable warp scheduler introduces $2^{32}$ bits-registers, and some bit comparison logics.) and no training process is needed. Without the training process, our programmable warp scheduler could achieve 6.9% of performance improvement over their warp scheduling techniques.

Gebhar et al. [33] proposed a short term warp scheduling technique called two-level warp scheduling. Their warp scheduler chooses the warps in the active warp groups. When a warp is stalled by long latency operations, it will be switched out from the active warp group. New active warps can be selected in a round robin manner or a greedy manner. The frequent change of the active warp group breaks intra-warp and inter-warp locality, which limits the performance gain. Compared to Gebhar’s two-level warp scheduler, our programmable warp scheduler improves the performance by 51.6%.

Narasiman et al. [47] proposed a two-level warp scheduling algorithm similar to Gebhar’s work [33]. Their scheduling techniques group nearby warps into warp groups of fixed size. The scheduler chooses one of the warp groups as high priority warp group.
When all the warps are stalled in the high priority warp group, another warp group is chosen as the high priority warp group. So to a certain extent, their scheduling technique will preserve inter-warp locality. The lifetime of the active warp group is still not long enough to maintain intra-warp locality. Compared to Gebhar’s and Narasiman’s scheduling techniques, the life time of the high priority warp group in our proposed programmable warp scheduler would be controlled by the intra-warp and inter-warp data reuse analysis based on the polyhedra model. Compared to Narasiman’s work, our programmable warp scheduler could obtain 49.2% of performance improvement.

Meng et al. [45] introduce dynamic warp subdivision (DWS) algorithm that further splits warps into smaller scheduling unit. Their technique allows SMs to fetch and issue instructions on different branches simultaneously. Their algorithm will not reduce the total number of warps competing for resources simultaneously, so it may make the situation even worse under certain conditions, since subdivision may combine uncoalesced global memory accesses in different branches.

Jablin et al. [37] introduce warp aware trace scheduling technique which optimizes the instruction level parallelism (ILP) of GPU kernels. Control flow graph and data flow graph are used to reschedule memory load instructions of GPU kernels. 10% performance gain on average can be achieved by taking advantage of ILP and data prefetching. Compared to their work, our work focuses on thread level parallelism and data reuses in L1 data cache.

Polyhedron model has been widely used in compiler optimization [10, 57, 6, 29, 30, 31]. Feautrier et al. [29, 30, 31] proposed the basic techniques related to the construction of polyhedron model and dependency graph for serial loops. We extend their polyhedron model to analyze the intra-warp and inter-warp data reuses of the input GPU kernels with the parallel features of the GPU kernels considered. Uday et al. [10] proposed an automatic parallelization framework called PLUTO for serial loops. In their work, they used data dependency graph based on polyhedron model to represent data dependency relationship. We extend their data dependency parsing module to analyze intra-warp, and inter-warp data.
4.7 Summary

In this chapter, we developed a compiler-assisted programmable warp scheduler that can take advantage of the intra-warp locality and inter-warp locality simultaneously. The intra-warp and inter-warp data reuse analysis is based on the modified polyhedron model for GPU kernels. Two extra PTX instructions, “setPriority” and “clearPriority”, are designed to support the programmable warp scheduler. A source to source kernel compiler pass is designed to do intra-warp, and inter-warp data reuse analysis, decide the best warp group size, and insert specific scheduler control instructions at proper positions automatically. Experimental results show that our programmable warp scheduler can achieve better performance compared to the existing hardware scheduling techniques (Our programmable warp scheduler could improve the performance of the input GPU programs by 6.9% on average over the CCWS. Our programmable warp scheduler could improve the performance of the input GPU programs by 51.6% on average over the Two-level scheduling technique proposed by Narasiman et al. [47]. Our programmable warp scheduler could improve the performance of the input GPU programs by 49.2% on average over the GTO scheduler proposed by Narasiman et al. [47]. The hardware overhead of our programmable warp scheduler is much smaller compared to CCWS, which needs the victim cache. Our compiler-assisted programmable warp scheduler just needs two extra registers and some bit computation logics.
Chapter 5: A Compiler-assisted CTA Mapping Scheme

5.1 Overview

When an SM has available CTA slots, a thread block will be selected and mapped to the SM in a round-robin manner [40]. The selection process does not consider the possible data locality or data reuses among the threads blocks mapped to the SM. Figure 5.1 shows a possible mapping result of a general scenario for a 1D applications. Assume the inter-CTA data locality exists among consecutive CTAs, the CTA mapping scheme shown in Figure 5.1 breaks the data locality, which increases the L1 cache footprint and degrades the overall performance.

![Figure 5.1: The default CTA mapping for 1D applications](image-url)
Figure 5.2: The CTA mapping. (a) Original mapping. (b) Mapping along the $x$ direction. (c) Mapping along the $y$ direction.

The same problem can be observed for the 2D applications too. Assume we have four SMs and a grid of $5 \times 5$ thread blocks. The thread block mapping pattern with the original mapping strategy will break inter thread block data reuses along the $x$ direction or the $y$ direction as illustrated in Figure 5.2(a). The inter thread block data reuses can be preserved if we map the CTAs along the $x$ direction as illustrated in Figure 5.2(b) because the CTAs having inter-warp data locality are all mapped to the same SM. Similarly, we can preserve the inter thread block data reuses along the $y$ direction by mapping the CTAs along the $y$ direction as illustrated in Figure 5.2(c).

In Chapter 4, we analyze the intra-warp and inter-warp data reuses in the L1 data cache. The analysis is based on an assumption that the number of the warps is limited in the same thread block. However, in the real GPU systems the threads are grouped and executed as thread blocks or CTAs. The best size of the high priority warp group can be larger than the number of warps in a single thread block. If we prioritize the warps from different thread blocks without considering the data locality among them, the data locality might be broken and the overall performance would be degraded. To construct the
high priority warp groups with the warps coming from the thread blocks with inter-CTA data reuses we must control the CTA mapping pattern according to the data reuse pattern among the thread blocks in the same kernel grid.

In [40], Lee et al. proposed a block CTA scheduling algorithm to preserve the inter-CTA locality. The block CTA scheduling algorithm can assign consecutive CTAs along the \( x \) direction to the same SM. However, without the inter-CTA reuse pattern detection mechanism, the algorithm cannot handle the inter-CTA reuses along the \( y \) direction. In addition, the algorithm cannot handle 2D applications.

In this chapter, we present a compiler-assisted locality aware CTA mapping scheme that can enable the users to set the CTA mapping scheme before a GPU kernel is launched. The compiler-assisted CTA mapping scheme based on the polyhedron model can detect and control the CTA mapping pattern automatically. Then, we combine the CTA mapping scheme with the compiler-assisted programmable warp scheduler to further improve the performance of the GPU kernels.

The rest of this chapter is organized as follows: in Section 5.2, we present the compiler-assisted locality aware CTA mapping scheme to detect the CTA mapping pattern. In Section 5.3, we illustrate how to combine the compiler-assisted CTA mapping scheme with the programmable warp scheduler. In Section 5.5, we evaluate the compiler-assisted locality aware CTA mapping scheme and the performance of the input benchmarks when they are optimized by the compiler-assisted CTA mapping scheme combined with the programmable warp scheduler. We conclude this chapter in Section 5.7.

### 5.2 The CTA Mapping Pattern Detection

We design a CTA mapping control API to modify the CTA mapping strategy before a kernel is launched. The API will modify the value of a special register that controls the CTA mapping unit of each SM core. For performance-complexity trade-off, we only consider
the three most common CTA mapping strategies: mapping along the $x$ direction, mapping along the $y$ direction and mapping in the round-robin manner. The default CTA mapping strategy is the round-robin mapping strategy.

Based on the CTA mapping API, we design a compiler-assisted locality aware CTA mapping scheme to insert the CTA mapping control APIs automatically based on the inter-CTA data reuse analysis. The basic rules of the compiler-assisted locality aware CTA mapping scheme is:

1. If there are inter thread block data reuses along the $x$ direction, then the CTAs are mapped along the $x$ direction.

2. If there are inter thread block data reuses along the $y$ direction, then the CTAs are mapped along the $y$ direction.

3. If there are inter thread block data reuses along both the $x$ and the $y$ directions, the CTAs are mapped along the direction that has larger reuse distance as the memory blocks can have more reuses along the direction having larger data reuse distance. For example, if the memory blocks accessed by a thread block can be reused by the following $N$ thread blocks, then these memory blocks can be reused $N - 1$ times during the execution of the GPU kernel. So the larger the $N$ is, the more times the memory blocks can be reused.

4. Otherwise, use the round-robin CTA mapping strategy.

The inter thread block data reuses along the $x$ direction can be formally defined as

**Definition 16. Inter-CTA data reuses along the $x$ direction** During the execution process of a thread block, if there exists two memory accesses $M$ and $M'$ that meet all of the following conditions, then inter-CTA data reuses exist among the thread blocks along the $x$ direction.
1. $M$ and $M'$ are issued by different thread blocks with the same thread block index in the $y$ direction.

2. $M'$ reuses the data in the L1 cache brought in by $M$.

The first constraint can be represented as

$$
\begin{align*}
0 & \leq bidy, bidy' < gdimy \\
0 & \leq bidx, bidx' < gdimx \\
0 & \leq tidy, tidy' < bdimy \\
0 & \leq tidx, tidx' < bdimx \\
bidy' &= bidy
\end{align*}
$$

(5.1)

The second constraint can be represented as

$$
\begin{align*}
\tilde{\alpha}'_i &= \tilde{\alpha}_i, i = 1, \ldots, m - 2 \\
\tilde{\alpha}'_{m-1} &= \beta' \ast B + \gamma' \\
\tilde{\alpha}_{m-1} &= \beta \ast B + \gamma \\
\beta' &= \beta \\
-B + 1 &\leq \gamma' - \gamma \leq B - 1 \\
0 &\leq \gamma' \leq B - 1 \\
0 &\leq \gamma \leq B - 1 \\
0 &\leq \beta' \\
0 &\leq \beta
\end{align*}
$$

(5.2)

Where $B$ indicates the cache block size; $\beta'$ and $\beta$ indicate the cache block indexes of $M'$ and $M$; $\gamma'$ and $\gamma$ indicate the offsets inside each cache block.
The target parameter that we are interested in is the reuse distance along the $x$ direction:

$$\zeta_x = |\text{bidx}' - \text{bidx}|$$ \hspace{1cm} (5.3)

Then, the problem of detecting inter thread block data reuses can be transformed into an integer linear programming problem:

$$\begin{align*}
\text{max} & \quad \zeta_x \\
\text{s.t.} & \quad \zeta_x = |\text{bidx}' - \text{bidx}| \\
& \quad \text{bidx}', \text{bidx} \in G_x
\end{align*}$$ \hspace{1cm} (5.4)

Where $G_x$ is the polyhedron defined by (5.1) and (5.2). If the problem (5.4) has no solution or $\zeta_x = 0$, then it indicates that there is no inter thread block data reuses along the $x$ direction.

Similarly, we can define inter-CTA data reuses along the $y$ direction as

**Definition 17. Inter-CTA data reuses along the $y$ direction** During the execution process of a thread block, if there exists two memory accesses $M$ and $M'$ that meet all of the following conditions, then inter-CTA data reuses exist among the thread blocks along the $y$ direction.

1. $M$ and $M'$ are issued by different thread blocks with the same thread block index in the $x$ direction.

2. $M'$ reuses the data in the L1 cache brought in by $M$.

The first constraint is represented as
The second constraint is the same as (5.2). We can obtain the maximum inter thread block reuse distance along the $y$ direction $\zeta_y$ in a similar way:

$$\begin{align*}
0 & \leq \text{bidx}, \text{bidx}' < \text{bdimx} \\
0 & \leq \text{tidx}, \text{tidx}' < \text{bdimx} \\
\text{bidx}' & = \text{bidx}
\end{align*} $$

(5.5)

(5.6)

Algorithm 2: Detect the CTA mapping direction

**Input:** GPU kernel $G$

**Output:** $direction$, $maxResueDistance$

1. $max_x = 0$
2. $max_y = 0$
3. **for** each memory access pair $M$ and $M'$ in $G$ **do**
   4. Get the reuse distance $r_x$ along the $x$ direction by solving problem 5.4;
   5. Get the reuse distance $r_y$ along the $y$ direction by solving problem 5.6;
   6. $max_x = \max(max_x, r_x)$;
   7. $max_y = \max(max_y, r_y)$;
4. **end**
5. **if** $max_x == 0$ and $max_y == 0$ **then**
6. $direction$ = round-robin;
7. **else**
8. $direction$ = $max_y > max_x$ ? $y : x$;
9. $maxResueDistance$ = $\max(max_y, max_x)$;
10. **end**

Algorithm 2 is presented to detect the CTA mapping direction. As illustrated in Al-
Algorithm 2, if there are multiple inter-CTA data reuses existing in a single GPU kernel, we record the maximum inter-CTA reuse distance along the $x$ direction and the $y$ direction separately (Line 1 – 5). When there are inter thread block data reuses along both the $x$ direction and the $y$ direction, the CTAs are mapped along the direction that has larger reuse distance (Line 6 – 9). Otherwise, the round-robin CTA mapping strategy would be used.

### 5.3 Combine the Programmable Warp Scheduler and the Locality Aware CTA Mapping Scheme

To take advantage of the inter thread block data locality and inter-warp data locality simultaneously, we combine the compiler-assisted programmable warp scheduler and the compiler-assisted locality aware CTA mapping scheme as illustrated in Algorithm 3.

**Algorithm 3:** Combine the CTA mapping scheme and the programmable warp scheduler

**Input:** GPU kernel $G$

**Output:** Optimized GPU kernel with both CTA mapping and programmable warp scheduler applied

1. Get CTA mapping direction through Algorithm 2;
2. Insert CTA mapping control API before $G$ is launched;
3. for each loop $L$ in $G$ do
4.  Apply Algorithm 1 on $L$ with the constrains modified by (5.7) and (5.8) to insert scheduler control instructions for $L$.
end

The compiler-assisted locality aware CTA mapping scheme will not affect the intra-warp data reuse detection algorithm. However, we must modify the inter-warp data reuse detection algorithm and the high priority warp group size detection algorithm. The constraints in (4.10) and (4.16) are modified by the extra constraints presented in (5.7) and (5.8). Also, we must remove the constraints of $w, w' < \text{ceiling}(bdimx * bdimy/32)$ because the warp IDs could exceed the thread block boundary. In addition, the check condi-
tion in Line 14 of Algorithm 1 must be modified to

\[ \min(\xi_{\text{min}}, \theta_{\text{min}}) < \#\text{warpsInBlock} \times \text{maxResueDistance}. \]

\[
\begin{cases}
  \text{bidy}' = \text{bidy} & \text{If CTAs are mapped along the } x \text{ direction} \\
  \text{bidx}' = \text{bidx} & \text{If CTAs are mapped along the } y \text{ direction}
\end{cases}
\]  

(5.7)

\[
\begin{cases}
  32w \leq \text{bdim}_x \times \text{tidy} + \text{tidx} + (\text{gdim}_x \times \text{bidy} + \text{bidx}) \times \text{wpb} < 32(w + 1) & \text{If CTAs are mapped along the } x \text{ direction or in the round-robin manner} \\
  32w \leq \text{bdim}_x \times \text{tidy} + \text{tidx} + (\text{gdim}_y \times \text{bidx} + \text{bidy}) \times \text{wpb} < 32(w + 1) & \text{If CTAs are mapped along the } y \text{ direction}
\end{cases}
\]  

(5.8)

Where \( \text{wpb} = \text{ceiling}(\text{bdim}_x \times \text{bdim}_y/32) \), which is the number of warps per thread block. The warp constraints for \( w' \) is modified in the same way.
Algorithm 4: CTA mapping

**Input:** gdimx, gdimy, #sms, direction

**Output:** the CTAs mapped to each SM

1. If direction == x then
   2. \[ a = \text{gdimy} / \#\text{sms}; \]
   3. \[ b = a \times \text{gdimx}; \]
   4. \[ c = (\text{gdimy} - a) \times \text{gdimx}; \]
   5. \[ d = (c + 1) / \#\text{sms} + 1; \]
   6. For each \( B = \text{CTA}(x, y) \) do
      7. If \( y < b \) then
         8. Assign \( B \) to SM core \( y / a \);
         end
      9. \[ e = (x + y \times \text{gdimx}) - b; \]
     10. Assign \( B \) to the SM core \( e / d \);
    end
   else if direction == y then
      11. \[ a = \text{gdimx} / \#\text{sms}; \]
      12. \[ b = a \times \text{gdimy}; \]
      13. \[ c = (\text{gdimx} - a) \times \text{gdimy}; \]
      14. \[ d = (c + 1) / \#\text{sms} + 1; \]
      15. For each \( B = \text{CTA}(x, y) \) do
         16. If \( x < b \) then
            17. Assign \( B \) to the SM core \( x / a \);
            end
         18. \[ e = (y + x \times \text{gdimy}) - b; \]
        19. Assign \( B \) to the SM core \( e / d \);
        end
   end
   else
      21. Assign the CTAs to the SMs in the round-robin manner;
   end
5.4 Balance the CTAs Among SMs

We modify the CTA mapping units in GPGPU-sim to evaluate our compiler-assisted locality aware CTA mapping algorithm. The CTA mapping unit can be configured to map CTAs along the \(x\) direction, along the \(y\) direction or using the round-robin manner. The CTA mapping configuration can be set by the CTA mapping control API we developed, which is supported by the GPGPU-sim modified by us. The CTA mapping control API can be executed before the GPU kernel launch. To balance the number of CTAs assigned to an SM, and preserve the inter-CTA data reuses as much as possible when the mapping direction is selected, we use the following rules

1. Assign a whole row or column of CTAs to an SM as long as we can evenly distribute them among the SMs.

2. For the rest of the CTAs, we evenly distribute them along the \(x\) direction or the \(y\) direction.

The CTA mapping algorithm that follows these two rules is illustrated in Algorithm 4. In Algorithm 4, \(a\), \(b\), \(c\), \(d\) and \(e\) are intermediate variables, \(direction\) is the CTA mapping direction that is detected by Algorithm 2. Figure 5.3 shows the mapping results of Algorithm 4 with \(#sms = 4\) for mapping along both the \(x\) direction and the \(y\) direction.

5.5 Evaluation

5.5.1 Evaluation Platform

The basic hardware configuration is the same as we described in Table 4.1. The benchmarks we selected to evaluate our CTA mapping algorithm are listed as follows:

(1) micro-benchmark: A simple GPU kernel designed to add neighboring blocks together
Figure 5.4: Micro benchmark as illustrated in Figure 5.4. We can see that the same block in input will be reused among CTAs along the \( x \) direction.

(2) matmul: Block matrix multiplication kernel, whose input size is 4kx4k.

(3) conv: Two dimensional convolution algorithm, whose input size is 4kx4k.

(4) demosaic: Image demosaicing algorithm, whose input size is 8kx8k.

(5) imregionmax: The algorithm to find the regional maximum for an image, whose input size is 4kx4k.

(6) tr: Matrix transpose algorithm, whose input size is 4kx4k.

(7) mum: Parallel local-sequence alignment program [5], whose input size is 50k.

(8) lps: A Laplace discretisation on a 3D structured grid [5], whose input size is 4M.

### 5.5.2 Experimental Results

To evaluate our compiler-assisted locality aware CTA mapping algorithm, we measure the performance of the GPU kernels optimized by the compiler-assisted locality aware CTA mapping algorithm (labeled as “cm” in Figure 5.5) and the performance of the GPU kernels optimized by the CTA mapping algorithm combined with the programmable warp scheduler discussed in Chapter 4 (labeled as “cm+ps” in Figure 5.5). We compare the performance of the GPU kernels optimized by our compiler-assisted locality aware CTA
mapping algorithm with the performance of the GPU kernels optimized by the GTO warp scheduler which is used as the baseline, the performance of the GPU kernels optimized by the CCWS warp scheduling algorithm proposed by Rogers et al. [51] and the performance of the GPU kernels optimized by the compiler-assisted programmable warp scheduler (labeled as “ps” in Figure 5.5).

The performance results are summarized in Figure 5.5 and the corresponding L1 cache miss rates of the GPU kernels are reported in Figure 5.6. The higher the speedup and the lower the L1 cache miss rate, the better the overall performance is. In Figure 5.5, the first column shows the normalized performance of the input benchmarks optimized by the GTO
algorithm. The second column shows the normalized speedups of the input benchmarks optimized by the CCWS warp scheduling algorithm. The third column shows the normalized speedups of the input benchmarks optimized by the compiler-assisted programmable warp scheduler. The fourth column shows the normalized speedups of the input benchmarks optimized by the compiler-assisted CTA mapping algorithm. The fifth column shows the normalized speedups of the input benchmarks optimized by the compiler-assisted CTA mapping algorithm combined with the programmable warp scheduler.

In Figure 5.6, the first column shows the normalized L1 cache miss rates of the input benchmarks optimized by the GTO algorithm. The second column shows the normalized L1 cache miss rates of the input benchmarks optimized by the CCWS warp scheduling algorithm. The third column shows the L1 cache miss rates of the input benchmarks optimized by the compiler-assisted programmable warp scheduler. The fourth column shows the L1 cache miss rates of the input benchmarks optimized by the compiler-assisted CTA mapping algorithm. The fifth column shows the L1 cache miss rates of the input benchmarks optimized by the compiler-assisted CTA mapping algorithm combined with the programmable warp scheduler.

As illustrated in Figure 5.5, our compiler-assisted locality aware CTA mapping algorithm can improve the performance of the selected benchmarks by 23.3% on average over the GTO warp scheduling algorithm. The GTO warp scheduler just considered the locality within each CTA, however, the inter thread block data reuses are ignored due to the round-robin CTA mapping strategy that always assigns adjacent CTAs to different SMs. According to Figure 5.6, our CTA mapping algorithm can reduce the L1 cache miss rates of the selected benchmarks by 18.9% on average compared to the GTO algorithm, which supports our previous discussions.

Compared to the GTO warp scheduling algorithm, the CCWS warp scheduling algorithm and the compiler-assisted programmable warp scheduler can avoid self-evictions in the L1 cache and preserve intra-warp and inter-warp data reuses. The CCWS warp schedul-
ing algorithm and compiler-assisted programmable warp scheduler improve the overall performance of the input benchmarks by 34.1% and 35.0 respectively on average over the GTO warp scheduling algorithm. However, The CCWS warp scheduling algorithm and the compiler-assisted programmable warp scheduler do not consider the data reuses among different CTAs, so they cannot further exploit the data reuses in the L1 cache. In Figure 5.6, we can see that the L1 cache miss rates of the input benchmarks are reduced by the CCWS warp scheduling algorithm and the compiler-assisted programmable warp scheduler by 12.0% and 12.7% respectively on average, which are smaller than the L1 cache miss rates decreased by the compiler-assisted CTA mapping algorithm.

The performance of the input benchmarks is improved by 56.7% on average when our compiler-assisted CTA mapping algorithm is combined with the compiler-assisted programmable warp scheduler, since the self-evictions in the L1 cache are avoided by limiting the total number of active warps and taking advantage of the inter CTA data reuses. In addition, our compiler-assisted CTA mapping algorithm can construct high priority warp groups across the CTA boundaries, which can completely hide long latency operations by feeding the warp scheduler with enough warps and taking advantage of the high data reuses in the L1 cache. This conclusion can also be supported by the reduced L1 cache miss rates of the input benchmarks optimized by the compiler-assisted CTA mapping algorithm combined with the programmable warp scheduler. The L1 cache miss rates of the input benchmarks optimized by the compiler-assisted CTA mapping algorithm combined with the programmable warp scheduler are reduced by 29.3% on average.

The benchmarks matmul, conv, demosaic, imregionmax and mum have both good intra-CTA and inter-CTA data locality, so the compiler-assisted programmable warp scheduler contributes most to the performance improvements (64% for matmul, 79% for conv, 55% for demosaic, 66% for imregionmax and 97% for mum) for these input benchmarks. While the rest of the benchmarks mainly obtained performance gain from the inter-CTA data reuses (100% for micro_bench, 99% for tr and 82% for lps). Both the compiler-assisted
programmable warp scheduler and the CCWS warp scheduler cannot further improve the performance of these input benchmarks.

From Figure 5.5 and Figure 5.6, we can also see that the performance gained from the L1 cache optimization is also different. For example, the combined scheme (the compiler-assisted CTA mapping algorithm combined with the programmable warp scheduler) reduces the L1 cache misses of the input benchmark demosaic by 25%. In contrast, the combined scheme reduces the L1 cache misses of the input benchmark lps by 36%. However, the performance of demosaic is increased by the combined scheme by 50%, higher than lps. The reason is that the ratios between the calculation instructions and the memory access instructions of different benchmarks are different. When the memory accesses of the benchmarks are the bottleneck, such as in the benchmarks conv, demosaic, imregionmax and micro_bench, significant performance improvement can be gained by decreasing the L1 cache miss rates. The performance improvement of the input benchmarks can achieve 89.5% on average from the L1 cache miss reduction of 32.2%. However, for the rest of the benchmarks, the performance improvement can only reach 23.9% from the L1 cache miss reduction of 23.9%.

5.6 Related Work

In [41], Lee et al. proposed two optimized CTA scheduling algorithms to improve the performance of the input GPU kernels. The lazy CTA scheduling method can limit the total number of active CTAs assigned to each SM to improve the L1 cache hit ratio and reduce the resource competition, which is similar to the idea of the warp limiting algorithm introduced by the CCWS warp scheduler [51] when applied on thread blocks. They also proposed the block CTA scheduling strategy to take advantage of the data locality among different thread blocks. Compared to their work, we propose a compiler-assisted locality aware CTA mapping algorithm which uses a systematic way to control the CTA
mapping pattern more accurately using the data reuse analysis based on the polyhedron model. In addition, the CTA limiting can also be performed in our framework by combining the compiler-assisted locality aware CTA mapping algorithm with the programmable warp scheduler, which can tune the number of active warps across the CTA boundaries.

Our compiler-assisted CTA mapping algorithm is also combined with the compiler-assisted programmable warp scheduler to exploit inter-CTA data reuses in addition to the intra-warp data reuses and the inter-warp data reuses, which exhibits more data reuses compared to the compiler-assisted programmable warp scheduling algorithms, such as the CCWS warp scheduling algorithm proposed by Rogers et al. [51] and the two-level warp scheduling algorithm proposed by Narasiman et al. [47]. Both of these two warp scheduling algorithms considered the data locality and the resource competition in the L1 cache among different warps on an SM, which can improve the overall performance by limiting the number of concurrent warps. However, just as we analyzed in this chapter, random CTA assignment to SMs might break the data locality and increase the L1 cache thrashing. Our proposed compiler-assisted CTA mapping scheme overcomes this drawback and further improves the overall performance.

In [38], the consecutive CTA mapping strategy is used to facilitate the design of optimized GPU memory prefetcher based on the two level warp scheduler proposed in [47]. Their prefetcher takes advantage of the spatial locality among consecutive CTAs assigned to each SM. However, only the benefit obtained from this CTA mapping strategy is analyzed and no inter-CTA data reuse analysis has been performed.

Yang et al. proposed a task scheduling algorithm considering data reuses in the cache, memory footprint and cache coherence for chip-multiprocessor systems [60]. They group tasks that have the maximum amount of data sharing into sharing groups. These tasks will be assigned to the same CPU core to maximize the data reuses in the L1 cache and minimize the cache coherence traffic, which is similar to the idea of our compiler-assisted CTA mapping algorithm that also assigns the thread blocks with data reuses to the same
SM. The concept to enable the memory footprint fit in the shared cache is also similar to the warp limiting technique used in our compiler-assisted programmable warp scheduler. However, they did not apply the data reuse analysis and memory footprint estimation at compile time.

5.7 Summary

In this chapter, we design a compiler-assisted locality aware CTA mapping scheme for GPUs to take advantage of the inter CTA data reuses in the GPU kernels. Using the data reuse analysis based on the polyhedron model, we can detect inter CTA data reuse patterns in the GPU kernels and control the CTA mapping pattern to improve the data locality on each SM. The compiler-assisted locality aware CTA mapping scheme can also be combined with the programmable warp scheduler to further improve the performance. The experimental results show that our CTA mapping algorithm can improve the overall performance of the input GPU programs by 23.3% on average and by 56.7% when combined with the programmable warp scheduler. In addition, the overall L1 cache misses can be reduced by 18.9% and 29.3% respectively by these two schemes.
Chapter 6: A Synchronization Optimization Framework for GPU kernels

6.1 Overview

As the power consumption and semiconductor technology are limiting the clock frequency increase, multi-core and many-core processors have become the major-trend of the new generation of processors [22, 35]. General purpose GPU is an effective many-core architecture, which has been used as the platform for many applications due to their powerful computation ability and massively parallel features [52, 11]. A well optimized GPU application program can achieve tens or even hundreds times of speedup compared to the same application program that runs on a single core CPU [26, 11, 43].

Barrier synchronizations are needed in GPU kernels to coordinate the execution of the threads in a thread block. In GPU kernels, the barrier synchronization is performed by inserting `syncthreads()` into the kernel programs [9] manually by the programmers, which is very labor intensive, and error-prone. Also, writing an optimized GPU program requires the programmers to be familiar with the hardware architectures and execution mechanisms of GPUs, which makes GPU programming very challenging for ordinary pro-
grammers [4, 61, 63, 27]. If a compiler framework can insert synchronization statements into GPU kernels automatically at compile time, it can reduce the workload of the GPU kernel programmers, and make parallel programming for GPU architectures easier.

In addition, barrier synchronization can be expensive in terms of execution time, because all the threads in a thread block need to wait at the synchronization point until all the threads reach this point. Synchronization also introduces extra warp switching, which can reduce the overall performance of GPU kernels. The timing performance of a commonly used GPU kernel, matrix multiplication, running on a GTX480 GPU with different number of synchronizations inserted, is shown in Figure 6.1. We can see that when the number of synchronizations increases, the total running time increases accordingly. So it is important to optimize the GPU kernels to reduce the number of synchronizations.

Several compiler frameworks have been proposed to do automatic GPU kernel optimization [61, 56, 4]. Yang et al. [61] proposed a compiler framework that can apply global memory access coalescing automatically. Van den Braak et al. proposed a solution that can take advantage of coalesced global memory accesses automatically via loop interchanging [56]. Auerbach et al. proposed a new compiler framework that can translate high level program annotations into CUDA kernels automatically in [4]. However, these compiler
frameworks adopted an approach to insert synchronizations using the most conservative strategy, which did not minimize the number of the synchronization statements.

In this chapter, we describe a synchronization optimization framework to automatically insert synchronization statements into GPU kernels at compile time, while minimizing the number of inserted synchronization statements.

The simplified version of a real GPU kernel in the benchmark `mv` shown in Figure 6.2(a) (The details of this GPU kernel is given in Section 6.4.1), is used to show our synchronization optimization framework can correctly insert the synchronization statements, and also minimize the number of inserted statements.

In Figure 6.3, ‘`idx`’ represents the global thread index along `x` direction. ‘`idy`’ represents the global thread index along `y` direction. Relative thread indexes inside thread blocks are represented as ‘`tidx`’ along the `x` direction and ‘`tidy`’ along the `y` direction. `shared_0` and `shared_1` are two arrays located in the shared memory. `A` and `B` are arrays located in the global memory. The corresponding correct output GPU kernel program with synchronization statements automatically inserted and minimized by our synchronization optimization framework is shown in Figure 6.2(b).

The rest of this chapter is organized as follows: In Section 6.2, we discuss data dependency analysis and derive the basic rules of synchronization insertion. In Section 6.3, we illustrate our synchronization optimization framework, which is to automatically insert synchronization statements at compile time, meanwhile minimize the number of inserted synchronization statements. In Section 6.4, we present the experimental results. Section 6.6 concludes this chapter.
\begin{figure}[h]
\centering
\begin{tabular}{ll}
(a) Input kernel & \quad \begin{aligned}
01: & \text{for}\ (i=0;\ i<\ \text{WIDTH}_A;\ i=i+1)\{ \\
02: & \quad \text{shared}_1[tidy][i][tidx] = A(idy,idx); \\
03: & \quad \text{if}(tidx < 16)\{ \\
04: & \quad \quad \text{shared}_0[tidx] = B[idx]; \\
05: & \quad \} \\
06: & \quad a = \text{shared}_1[tidx][tidy][i]; \\
07: & \quad b = \text{shared}_0[i]; \\
08: & \} \\
\end{aligned} \\
(b) Output kernel with synchronizations inserted & \quad \begin{aligned}
01: & \text{for}\ (i=0;\ i<\ \text{WIDTH}_A;\ i=i+1)\{ \\
02: & \quad \_\_\text{syncthreads}(); \\
03: & \quad \text{shared}_1[tidy][i][tidx] = A(idy,idx); \\
04: & \quad \text{if}(tidx<16)\{ \\
05: & \quad \quad \text{shared}_0[tidx] = B[idx]; \\
06: & \quad \} \\
07: & \quad \_\_\text{syncthreads}(); \\
08: & \quad a = \text{shared}_1[tidx][tidy][i]; \\
09: & \quad b = \text{shared}_0[i]; \\
10: & \} \\
\end{aligned}
\end{tabular}
\caption{An example code segment}
\end{figure}
Figure 6.3: The CFG of the example code in Figure 6.2(a)
6.2 Basic Synchronization Insertion Rules

6.2.1 Data Dependencies

A data dependency exists between two statements $S_1$ and $S_2$ if there exists a feasible runtime execution path from $S_1$ to $S_2$, and both $S_1$ and $S_2$ access the same memory location, and at least one of the accesses is a write access. $S_1$ is defined as the source of the data dependency and $S_2$ as the sink of the data dependency. And the data dependency is denoted as $d = (\text{source, sink})$. Three types of data dependencies, true data dependencies, anti data dependencies, and output data dependencies, must be preserved when program transformation is applied.

A data dependency is defined as an intra-warp data dependency (IWD) if $S_2$ accesses the data accessed by $S_1$ in the same warp for the input GPU kernel program. Otherwise the data dependency is defined as a cross-warp data dependency (CWD).

When a CWD exists, synchronization statements must be inserted into the GPU kernel program to coordinate the execution of the threads. The execution process of a simple GPU kernel fragment without synchronization inserted between $S_1$ and $S_2$ in Figure 6.4 is used to illustrate the importance of synchronization statements when there are CWDs in the GPU kernels. For simplicity, assume there are two threads and the SIMD width is 1. $A[]$ is an array located in the shared memory. The initial value of each element in array $A$ is $x$. The red arrows in Figure 6.4 represent violated data dependencies. The red assignment statements of each subgraph represent the situation where variables get wrong values due to data dependency violation.

In Figure 6.4(a), only IWDs exist, so no data dependency is broken. In Figure 6.4(b) true CWDs exist. The variable $b$ in thread 0 gets wrong value because $A[1]$ has not been assigned by thread 1. Anti-CWDs are illustrated in Figure 6.4(c). Variable $b$ gets wrong value in thread 1 because $A[1]$ is modified by thread 0 before it is read by thread 1.
Figure 6.4: The execution process without synchronizations (We assume each warp has a single thread for illustration purpose)
Figure 6.4(d) output CWDs are illustrated. \( A[1] \) gets wrong value in thread 1 because thread 1 assigns a new value to it after it was already assigned the right value.

In an input GPU kernel, output dependencies only exist in a very special situation where no read accesses exist between two consecutive write accesses of the same memory location. In this situation, useless instructions are generated as illustrated in Figure 6.4(d). So we do not consider output dependencies in our future analysis.

To enforce the correct execution of the GPU kernel programs with CWDs, synchronization statements must be inserted. No synchronization is needed for IWDs because the data dependency source is always executed before the data dependency sink in the same thread as shown in Figure 6.4(a). Based on the analysis, we derive the first rule of synchronization insertion as follows:

**Rule 1.** At least one synchronization statement must be placed between the data dependency source and the data dependency sink for true CWDs or anti-CWDs. No synchronization statement is needed for IWDs.

The data dependency is preserved if a synchronization statement exists between the data dependency source and the sink of a CWD. So the second rule of synchronization placement is derived as follows to avoid redundant synchronization statements.

**Rule 2.** Redundant synchronizations between the data dependency source and the sink of a CWD are not necessary and must be removed.

At the same time, we must ensure the safety of the CWDs.

**Rule 3.** If a data dependency cannot be proved to be an IWD at compile time, it must be treated as a CWD.
6.2.2 Rules of Synchronization Placement with Complex Data Dependencies

In Section 6.2.1, we derived the rules to insert minimum number of synchronizations into a GPU kernel program without branches and loops, i.e., a program that can be represented as a basic block. In this section, we present the basic synchronization insertion rules for complex programs with branches and loops, in which multiple feasible execution paths might exist between multiple data dependency sources and multiple data dependency sinks.

In Figure 6.5(a), $M$ statements $T_1,T_2,...,T_M$ depend on $N$ dependency source statements $S_1,S_2,...,S_N$. There are totally $M \times N$ possible data dependencies $d_{i,j} = (S_i,T_j)$. A Proxy Source (PS) is placed at the beginning of the merge node of the paths from all the data dependency sources to the corresponding data dependency sinks to take over the role of those real data dependency sources. The PS representing the $N$ real data dependency sources acts like a real data dependency source. The dependency type of $d_j = (PS,T_j)$ can be determined by a very simple rule. When the data dependency between the real data dependency sources $S_i$ represented by the PS, and the data dependency sink $T_j$, $d_{i,j} = (S_i,T_j)$, is a CWD, then the data dependency $d_j = (PS,T_j)$ is a CWD. Other-
wise, the data dependency $d_j = (PS, T_j)$ is an IWD.

The algorithms to transform an input GPU kernel program into its SSA-like form are presented in Algorithm 5 and Algorithm 6, which are used to find the merge node, insert PSs, and identify data dependency sources, and sinks. The data dependencies shown in Figure 6.5(a) are transformed into the SSA-like form in Figure 6.5(b) using these algorithms.

For general programs with or without branches and loops represented in the SSA-like form, the synchronization placement rule is derived as follows.

**Rule 4.** Synchronization statements must be inserted immediately after each data dependency source or PS when the data dependency source or the PS is involved in one or more CWDs in the GPU kernel programs represented in the SSA-like form, so the CWDs are all preserved and the number of inserted statements is the minimum.

**Proof.**

1. Assume CWDs exist between a real dependency source $X$ and $M$ dependency sinks $T_1, T_2, ..., T_M$. If we put the synchronization statement immediately after $X$, all the CWDs $d_i = (X, T_i)$ are preserved, since $X$ dominates all the data dependency sinks $T_1, T_2, ..., T_M$ according to the definition of the SSA-like form. And the total number of synchronization statements inserted is 1, which is the minimum. However, if we put a synchronization statement immediately before $T_i$, or after an intermediate statement $P$ on the path from $X$ to $T_i$, to preserve the data dependency $d_i = (X, T_i)$, then the data dependency $d_j = (X, T_j)$ cannot be preserved by the inserted synchronization statement, since $T_i$ or $P$ may not dominate all the other data dependency sinks $T_1, T_2, ..., T_M$. So extra synchronization statements are needed, which will increase the total number of synchronization statements.

2. Assume $N \times M$ CWDs exist between $N$ data dependency sources $S_1, S_2, ..., S_N$ rep-
resented by a PS $X$, and the dependency sinks $T_1, T_2, \ldots, T_M$. According to the characteristics of the SSA from, all the paths from $N$ data dependency sources $S_1, S_2, \ldots, S_N$ to the $M$ data dependency sinks $T_1, T_2, \ldots, T_M$ merge at node $X$. Since the merge node $X$ dominates all the data dependency sinks $T_1, T_2, \ldots, T_M$, then all the data dependencies are preserved by the synchronization statement inserted at the beginning of the merge node $X$. The total number of synchronization statements inserted is 1, which is the minimum. Otherwise, if we put the synchronization statement on the path from a data dependency source $S_i$ to the merge node $X$, then it is not guaranteed that all the data dependencies can be preserved, and extra synchronization statements might be needed. In contrast, if we put the synchronization statement on the path from the merge node $X$ to $S_j$, again it is not guaranteed that all the data dependencies can be preserved, which would lead to inserting extra synchronization statements.

As shown in Figure 6.6, after transforming our CFG into the SSA-like form, the pivot
node can be a real source node or a PS. Assume all the data dependencies are CWDs, when we put synchronization on the pivot node we can reserve all of the data dependencies related to it. Only one synchronization is needed. But if we put synchronization somewhere else (Assume we put the synchronization on Node X as shown in Figure 6.6 for illustration purpose). We cannot cover all of the data dependencies. We may need more than one synchronization to cover all of the data dependencies.

6.3 Synchronization Optimization Framework

In Section 6.2, we derived basic synchronization insertion rules for GPU kernels represented in SSA-like form. We design our synchronization optimization framework based on those basic synchronization insertion rules, which is illustrated in Figure 6.7. The key components of our synchronization optimization framework are illustrated in the following subsections. As shown in Figure 6.7, the input CFG will be transformed into SSA-like form first by the transform algorithms illustrated in Section 6.3.1 and Section 6.3.2. The main part of our synchronization optimization framework, i.e., the loop inside the dashed box shown in Figure 6.7 is to determine where to insert synchronization statements.

We adopt the polyhedron model to test whether or not the data dependency between two memory accesses is a CWD if both of them are affine memory accesses. The algorithm to determine whether or not all the CWDs related to a specific statement have been synchronized is illustrated in Section 6.3.4. The code generation phase of the synchronization optimization framework is briefly introduced in Section 6.3.5.

To facilitate the implementation of our synchronization optimization framework, ‘sync’ flag is marked on statements during analysis phase indicating that synchronization statements need to be inserted after these statements in the code generation phase. ‘sync’ flag would only be put on the data dependency sources or PSs which fulfills the requirements of Rule 4. The requirements of Rule 2 are also fulfilled by marking ‘sync’ flags. No
Figure 6.7: The synchronization optimization framework
‘sync’ flag would be marked if no CWD exists as illustrated in Rule 1. We also design a CWD detection algorithm 6.3.3 to enforce Rule 3 based on the polyhedron model. In other words, our synchronization optimization framework will enforce all the basic synchronization rules we deduced in Section 6.2.

Inside the loop inside the dashed box in Figure 6.7, each statement with a data dependency source or a PS is analyzed. If a statement is involved in a CWD and at least one CWD related to this statement has not been synchronized already, a ‘sync’ flag will be put on this statement. After each round the statements with ‘sync’ flags will be stored and a new round will be started to repeat the synchronization candidate search process. The iterative process stops when the results of two continuous rounds are exactly the same, and the iterative process always converges.

Also, no redundant synchronizations will be introduced by our synchronization optimization framework. If a synchronization statement \( S \) is redundant, then it means all the CWDs involved can be preserved by other CWDs. In this situation, \( S \) would be removed according to our algorithm. The iterative process ensures that no redundant synchronizations will be inserted.

### 6.3.1 PS Insertion

We modified the original SSA transform algorithm to transform the input programs into their SSA-like forms \([50]\). The SSA-like form transform consists of two phases: (1)PS insertion (2) Classification of data dependency sources and sinks. We place PSs where the original SSA transform algorithm places phi-functions as illustrated in Algorithm 5.

As illustrated in Algorithm 5, PSs are inserted at the iterative dominance frontier of the real data dependency sources. The set \( \text{EverOnWorkList} \) holds the nodes that have already been analyzed, which is used to ensure that no redundant PSs are inserted.

In Algorithm 5, PSs are inserted independently for different types of data dependen-
Algorithm 5: Insert PSs

**Input:** CFG, Memory array $A$, and dependency type $T$
**Output:** CFG with PSs inserted

$WorkList = \emptyset$;
$EverOnWorkList = \emptyset$;
$AlreadyHasPS = \emptyset$;

for each node $N$ in the CFG containing sources of $T$ for $A$ do

$WorkList += N$;
$EverOnWorkList = WorkList$;

while $WorkList \neq \emptyset$ do

Remove a node $X$ from $WorkList$;

for each $d \in DF(X)$ do

if $d \notin AlreadyHasPS$ then

Insert PS at the beginning of $d$;
$AlreadyHasPS += d$;
if $d \notin EverOnWorkList$ then

$WorkList += d$;
$EverOnWorkList += d$;

end

end

end
cies. When we analyze true data dependencies, write accesses of array \( A \) are treated as data dependency sources and read accesses of array \( A \) are treated as data dependency sinks.

The green arrow in Figure 6.8 shows a special data dependency type, i.e., live out data dependency of a loop, which needs to be considered by our PS insertion algorithm. Live out data dependencies of a loop are data dependencies whose data dependency sources \( S_1 \) is the data dependency sources in a loop body and all the sinks depending on that data dependency source are outside the loop body.

In this chapter, the prologue node of a loop in a CFG is defined as the corresponding node of the condition test statement of a loop. When we insert synchronization statements following Rule 4 on PS located at the beginning of the prologue node, the synchronization statement executes \( N + 1 \) times where \( N \) is the total number of loop iterations. However, if we put the synchronization immediately after the loop body, the live out data dependency can also be preserved and the synchronization statement would execute just once. To minimize the execution time of the synchronization statements, we must process the execution path of a loop body and the execution path exiting the loop separately. The simplest way to achieve this goal is to insert two extra PSs at the beginning of the loop body and the node immediately following the loop as shown in Figure 6.8. After the two extra PSs are
Figure 6.9: The CFG with PSs inserted
inserted, synchronization statements are inserted after the PS which is at the node imme-
diately following the loop for live out data dependencies. And no more synchronizations
need to be placed in the prologue node of the loop because no data dependency sink can
see the prologue node anymore.

The CFG with PSs inserted according to Algorithm 5 for the input code in Fig-
ure 6.2(a) is shown in Figure 6.9. The PSs are inserted by analyzing the true data de-
pendencies regarding the access of array shared_0 as shown in Figure 6.9. Initially, node 7
is put into WorkList since Node 7 contains a write access of array shared_0. The first PS
is put in node 8 because $DF(7) = \{8\}$. Then node 8 is added into WorkList and a PS is
placed on node 3 because $DF(8) = \{3\}$. At this point, WorkList became empty, so the
phase 1 of the algorithm ends. Finally, two extra PSs are placed on node 1 and node 4 for
the live out data dependencies of the loop.

### 6.3.2 Classification of Data Dependency Sources and Sinks

After PSs are inserted in a CFG, the variable renaming algorithm [50] is modified to clas-
sify data dependency sources and sinks for a specific dependency type with regard to the
memory access of a particular array. The algorithm to classify the data dependency sources
and sinks is presented in Algorithm 6.

After the algorithm completes, each data dependency source or PS will get a unique
classId, which can be used to identify the data dependency sources and sinks. A data
dependency sink can only see the dependency source with the same classId.

In Algorithm 6, the ancestor set is used to record the classIds of the ancestors of
each PS, including the data dependency source and PSs that can reach this PS. Algorithm 6
is a recursive algorithm, and it is performed on node $X$ as well as on all the children of $X$
in the IDOM tree. Every statement in node $X$ is processed in order in Algorithm 6. If a
statement contains a data dependency sink, the classId of the data dependency sink is set
Algorithm 6: Classify Data Dependency Sources And Sinks

Input: CFG with PSs inserted, IDOM tree, memory accesses of a particular array
Output: Data dependency sources and sinks labeled with class id

stack idStack = ∅;
C = 0;
set all the classIds to 0;
initialize the ancestor set for all the PSs;
classifySearch(X:cfg node);
for each statement T in X do
    for each data dependency sink N in T do
        N.classID = idStack.top();
    end
    for each data dependency source S in T do
        S.classID = C;
idStack.push(C);
        C++;
    end
end
for Y in SUCC(X) do
    if there is a PS at the beginning of Y then
        PS.ancestors += idStack.top();
    end
end
for each children D of X in IDOM tree do
    call classifySearch(D);
end
for each data dependency source in X do
    idStack.pop();
end
Figure 6.10: The CFG with data dependency sources/sinks classified to current classId stored on the top of idStack. On the other hand, if a data dependency source is encountered, the next available classId denoted as variable $C$ in Algorithm 6 is used as the classId of the data dependency source. The value stored in variable $C$, the next available classId, is then pushed to the idStack stack. If there are PSs in SUCC(X), current classId on the top of idStack is added to the ancestor set of those PSs indicating that data dependency source with the same classId can reach them.

The CFG of the example code shown in Figure 6.2(a) with data dependency sources and sinks labeled by Algorithm 6 is shown in Figure 6.10. In Figure 6.10, data depen-
Sources: PS0, PS1, PS2, S3, PS4  
Sinks: N4

group 0: PS0, PS2

group 1: PS0, PS1

group 2: S3

group 3: PS2, S3, PS4, N4

group 4: PS4

Figure 6.11: The data dependency source and sink groups

dency sources and PSs with \textit{classId m} are denoted as \textit{Sm} and \textit{PSm}. Data dependency sinks depending on the data dependency source with \textit{classId m} are denoted as \textit{Nm}. The \textit{classIds} of the real data dependency sources or \textit{PSs} represented by \textit{PSm} are enclosed in the parentheses following \textit{PSm}.

After the data dependency sources and sinks are classified, five groups of data dependency sources and sinks are obtained as shown in Figure 6.11, which is in SSA-like form. With data dependencies represented in SSA-like form in Figure 6.11, we can apply Rule 4 to insert synchronizations for each data dependency group.

6.3.3 Identify IWDs and CWDs

In order to avoid inserting unnecessary synchronizations according to Rule 1, whether a data dependency is a CWD or an IWD must be determined automatically at compile time. According to the lock step execution manner of the threads in the same warp [9], CWDs between two threads in the same warp will be preserved automatically. So we do not need barrier synchronizations for these CWDs. We treat these CWDs inside the same warp as IWDs. CWDs can be formally defined as
Definition 18. **CWD** During the execution process of a thread block, if two memory accesses, \( M \) and \( M' \), meet all of the following conditions, then the data dependency between \( M \) and \( M' \) is a **CWD**.

1. The thread that issues \( M \) and the thread that issues \( M' \) are in different warps.
2. \( M' \) is issued later than \( M \).
3. \( M' \) and \( M \) access the same location in the shared memory.

Condition 1 in Definition 18 can be represented by inequalities:

\[
\begin{align*}
0 \leq \text{bidy}, \text{bidy}' &< \text{gdimy} \\
0 \leq \text{bidx}, \text{bidx}' &< \text{gdimx} \\
0 \leq \text{tidy}, \text{tidy}' &< \text{bdimy} \\
0 \leq \text{tidx}, \text{tidx}' &< \text{bdimx} \\
32w &\leq \text{bdimx} \cdot \text{tidy} + \text{tidx} < 32(w + 1) \\
32w' &\leq \text{bdimx} \cdot \text{tidy}' + \text{tidx}' < 32(w' + 1) \\
0 &\leq w, w' < \text{ceiling}(\text{bdimx} \cdot \text{bdimy}/32) \\
0 &< |w - w'|
\end{align*}
\]  

(6.1)

The second condition can be represented with the sequencing predicate [29] for each possible iteration depth \( p \).

\[
\begin{align*}
\vec{i}[0..p] = \vec{i}[0..p] \land \vec{i}[p + 1] > \vec{i}[p + 1] & \quad \text{when } 0 \leq p < N \\
\vec{i}[0..N - 1] = \vec{i}[0..N - 1] \land T_{M,M'} = \text{true} & \quad \text{when } p = N 
\end{align*}
\]  

(6.2)

Where
\[ \vec{i}' = \begin{pmatrix} i_0' & i_1' & \ldots & i_{N-1}' \end{pmatrix}^T \subseteq \vec{e}', \vec{i} = \begin{pmatrix} i_0 & i_1 & \ldots & i_{N-1} \end{pmatrix}^T \subseteq \vec{e} \]  

(6.3)

\[ T_{M,M'} \text{ is true when } M \text{ textually precedes } M'. \ T_{M',M} \text{ is false when } M' \text{ and } M \text{ are issued by the same statement.} \]

The third condition can be represented as:

\[
\begin{cases}
\alpha_1' = \alpha_1 \\
\alpha_2' = \alpha_2 \\
\ldots
\end{cases}
\]

(6.4)

Which means the corresponding subscripts of the two array accesses, M, and M’, are identical.

When the dependency is a CWD, the target parameter that we would be interested is the reuse distance \( \vec{i}' \) and \( \vec{i} \):

\[ \vec{\delta}_p = \vec{i}' - \vec{i} \]  

(6.5)

So the problem of the CWD detection at iteration level \( p \) can be represented as the following integer linear programming problem:

\[
\max_{\vec{\delta}_p} \quad \vec{\delta}_p
\]

s.t.  
\[ \vec{\delta}_p = \vec{i}' - \vec{i} \]  

(6.6)

\[ \vec{i}', \vec{i} \in D_p \]

\( D_p \) is the polyhedron defined by formula (6.1), (6.2), (6.4), and (3.7).

We will solve the problem of (6.6) for all of the iteration levels (i.e. the common loop levels). If one of them has a solution then the data dependency is a CWD. Otherwise, if none of them has a solution, the data dependency will be treated as an IWD.
If one of the array accesses of the data dependency is not affine, such as the indirect memory access $a[b[i]]$, we will treat it as a CWD according to Rule 3.

### 6.3.4 Existing Synchronization Detection

We need to check whether synchronizations already exist on all the execution paths from a specific data dependency source to a sink in order to avoid redundant synchronizations. Two possible situations should be considered while checking the existing synchronizations:

1. The data dependency source and the data dependency sink are in the same basic block and the data dependency source precedes the sink.
2. The data dependency source and the data dependency sink are inside different basic blocks or they are in the same basic block but the source is after the sink statically and there is a feasible backward execution path from the source to the sink. The intermediate nodes on the path from the source to the sink must be considered besides the data dependency source node (start node) and the sink node (end node).

For situation(1), if a statement between the data dependency source and the sink has been marked with a ‘sync’ flag, then the data dependency source and sink are already synchronized. For situation(2) the existing synchronizations can be detected in two steps.

Step1: Check if synchronizations exist after the data dependency source in the basic block where the source is located (start node). Check if synchronizations exist before the data dependency sink in the basic block where the sink is located (end node). Step2: If no ‘sync’ flags are found in step1, check if synchronizations exist in the intermediate nodes on all the possible execution paths from start node to the end node using Algorithm 7.

Three possible values, 0, 1, or 2, might be returned by Algorithm 7: (1). 0 denotes there is no execution path from a node $X$ to the end node. (2). 1 denotes that all the possible execution paths from a node $X$ to the end node are already synchronized. (3). 2 denotes that not all the possible execution paths from a node $X$ to the end node have
Algorithm 7: Search Intermediate Nodes

**Input:** CFG, the start node, and the end node of the CFG

**Output:** output: 0, 1 or 2

stack pathStack = ∅;

\( X = \text{start} \);

intermediateSearch(\( X \), and the end node of CFG) ;

if pathStack contains \( X \) or \( X \) is end then
  return 0;
end

pathStack.push(\( X \));

if \( X == \text{end} \) then
  \( T = \text{pathStack} . \text{clone}() \);
  result = 2;
  while \( T \) is not empty do
    \( Y = T . \text{top}() \);
    if (\( Y \) is not the start or the end) and (\( Y \) contains synchronizations) then
      result = 1;
      break;
    end
  end
else
  if \( \text{SUCC}(X) \) is \( \emptyset \) then
    result = 0;
  else
    result = 0;
    for each node \( M \) in \( \text{SUCC}(X) \) do
      \( r = \text{intermediateSearch}(M, \text{end}) \);
      if \( r \neq 0 \) then
        result = 1;
        if \( r == 2 \) then
          result = 2;
        end
      end
    end
  end
end

pathStack.pop();
return result;
been synchronized. During the recursive process of Algorithm 7, stack `pathStack` is used to record the nodes on the path from the `start` node to a node `X`. During each recursive call of Algorithm 7, node `X` is processed in three different ways: (1) If node `X` is a node already accessed or `X` is the exit node of the CFG, `intermediateSearch` returns 0. (2) If node `X` is the end node, and all the nodes on the path from `start` to `X` have been checked, and if synchronizations exist in one or more of these nodes, `intermediateSearch` returns 1. otherwise, 2 is returned. (3) IF node `X` is a node whose SUCC(X) is not empty, then `intermediateSearch` is called recursively for the nodes in SUCC(X). If all these recursive calls return 0, 0 is the final return value. If all the non-zero return values are 1, 1 is the final return value. Otherwise 2 is returned indicating that not all the paths from the `start` node to the `end` node going through node `X` are synchronized.

### 6.3.5 Code Generation

In this subsection, we briefly introduce the code generation phase of our synchronization optimization framework. When a ‘sync’ flag is marked on a real memory access statement, the ‘`syncthreads()`’ statement must be placed immediately after the memory access statement. Otherwise when a ‘sync’ flag is marked on a PS, the synchronization insertion rule is summarized as follows by checking the statement `K` immediately after the PS: 

(1) `K` is an initializing statement of a for-loop: put ‘`syncthreads()`’ immediately before the loop.

(2) `K` is an increment statement of a for-loop: put ‘`syncthreads()`’ at the end of the loop body.

(3) `K` is a conditional expression of an if-statement: put ‘`syncthreads()`’ immediately before the if-statement.

(4) `K` is a statement excluded by the above three conditions: put ‘`syncthreads()`’ immediately before `K`. 
As we discussed in Section 6.3.1, ‘sync’ flag would never be put on the prologue node of a loop, so the situation where ‘sync’ flags are on for-loop’s or while-loop’s condition statements is not considered in the code generation phase.

6.3.6 An Illustration Example

The working process of our synchronization optimization framework summarized in Figure 6.7 is illustrated in Figure 6.12 with the example code in Figure 6.2(a) as the input program. The SSA-like form representation of the example code in Figure 6.2(a) is shown
in Figure 6.10. After the first round of the searching process, three ‘sync’ flags are marked: The ‘sync’ flag on statement 1 of node 4 is set by the true CWD between statement 1 of node 4 and statement 1 of node 8 with regard to the memory accesses of shared_1. ‘sync’ flag on statement 0 of node 4 is set by the anti-CWD between statement 1 of node 8 and statement 1 of node 4 with regard to the memory accesses of shared_0. And the ‘sync’ flag on statement 0 of node 8 is set by the true CWD between statement 0 of node 7 and statement 2 of node 8 with regard to the memory accesses of shared_0.

In the second round of the searching process, the ‘sync’ flag on statement 1 of node 4 is removed because the data dependency between statement 1 of node 4 and statement 1 of node 8 is already preserved by the ‘sync’ flag on statement 0 of node 8. Finally, in the third iteration, no ‘sync’ flag is added or removed, the algorithm terminates with the final result CFG marked with ‘sync’ flags shown in Figure 6.12. We can see that the iterative process of the algorithm removes redundant synchronizations systematically.

6.4 Evaluation

6.4.1 Experimental Platform

Our synchronization optimization framework is implemented in the GPU compiler framework proposed by Yang Yi et al. in [61], which is based on CETUS, a source-to-source compiler framework [36]. This compiler framework can provide us an object-oriented intermediate representation (IR) and several useful compiler tools including CFG construction. When we need to detect whether or not a data dependency is a CWD, we will generate an intermediate source code and specify the source and sink of the data dependency. Then our polyhedron model based analyzer is called to do the test for us. Two kinds of benchmarks are used as inputs in our experiments to verify the correctness of our synchronization optimization framework, which are introduced briefly as follows:
1. The optimized GPU kernels generated by Yang Yi’s GPU compiler framework.

(1) matmul: Block matrix multiplication kernel, whose input size is 4Kx4K.
(2) mv: Matrix vector multiplication, whose input size is 4kx4k.
(3) tmv: Transposed matrix vector multiplication, whose input size is 4kx4k.
(4) transpose: Matrix transpose algorithm, whose input size is 4kx4k.
(5) conv: Two dimension convolution algorithm, whose input size is 4kx4k.
(6) imregionmax: The algorithm that can find the regional maximum for an image, whose input size is 4kx4k.
(7) reduction: The algorithm that performs the two-dimensional reduction operation, whose input size is 4kx4k.

2. The benchmarks obtained from Rodinia GPU benchmark suit [15], in which kernels are optimized manually.

(1) backprop: Back propagation algorithm used in the training process of layered neural networks, whose input is 65536 points.
(2) hotspot: A thermal simulation algorithm, whose input size is 512x512.
(3) kmeans: A cluster algorithm that is widely used in the field of data-mining, whose input is 494020 records.

Our synchronization optimization framework preprocesses the GPU kernels by removing all of the existing ‘_syncthreads()’ statements. Then our synchronization optimization framework automatically insert the synchronization statements into the preprocessed GPU kernels, while minimizing the number of inserted synchronization statements.

The correctness of our synchronization optimization framework is verified by comparing the execution results of the optimized output GPU kernels produced by our synchronization optimization framework, and the original GPU kernels. In rare cases, we also manually check the correctness of the optimized output GPU kernels produced by our synchronization optimization framework. The total number of synchronizations inserted and the number of synchronizations executed are compared between the optimized output GPU
6.4.2 Experimental Results

The total number of ‘_syncthreads()’ statements inserted (NS) and the total number of synchronizations executed in a thread (NE) for the original kernels and the optimized kernels by our synchronization optimization framework are reported and compared in Table 6.1. In Table 6.1, the number of synchronizations executed in each thread of the input kernel programs is recorded using a counter during execution.

From the comparison results presented in Table 6.1, we can see that our synchronization insertion algorithm places ‘_syncthreads()’ statements correctly, while minimizing the total number of synchronizations. Our synchronization optimization framework can remove redundant synchronizations in the loop body of the benchmarks, *mv*, *conv* and *backprop*. Thus, the total number of synchronizations executed is reduced significantly. For the benchmarks, *kmeans*, *imregionmax*, *hotspot*, *transpose*, and *reduction*, the redundant synchronization statements outside a loop are removed. So the total number of

<table>
<thead>
<tr>
<th>name</th>
<th>original</th>
<th></th>
<th>optimized</th>
<th></th>
<th>correct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NS</td>
<td>NE</td>
<td>NS</td>
<td>NE</td>
<td></td>
</tr>
<tr>
<td>matmul</td>
<td>2</td>
<td>512</td>
<td>2</td>
<td>512</td>
<td>yes</td>
</tr>
<tr>
<td>mv</td>
<td>4</td>
<td>1024</td>
<td>2</td>
<td>512</td>
<td>yes</td>
</tr>
<tr>
<td>tmv</td>
<td>2</td>
<td>256</td>
<td>2</td>
<td>256</td>
<td>yes</td>
</tr>
<tr>
<td>transpose</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>conv</td>
<td>4</td>
<td>256</td>
<td>2</td>
<td>128</td>
<td>yes</td>
</tr>
<tr>
<td>imregionmax</td>
<td>68</td>
<td>68</td>
<td>67</td>
<td>67</td>
<td>yes</td>
</tr>
<tr>
<td>reduction</td>
<td>17</td>
<td>17</td>
<td>7</td>
<td>7</td>
<td>yes</td>
</tr>
<tr>
<td>backprop</td>
<td>5</td>
<td>8</td>
<td>3</td>
<td>6</td>
<td>yes</td>
</tr>
<tr>
<td>hotspot</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>yes</td>
</tr>
<tr>
<td>kmeans</td>
<td>4</td>
<td>10</td>
<td>3</td>
<td>9</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 6.1: Comparisons of the number of synchronizations
synchronizations executed is still less than that of the original kernels. For the benchmarks, *matmul* and *tmv*, all the original synchronization statements are already in the optimized positions. The output kernels generated by our synchronization optimization framework are the same as the original kernels, which is correct, and the number of synchronization statements is minimized. To summarize, the number of synchronization statements in GPU kernels is reduced by 32.5%, and the number of synchronization statements executed is reduced by 28.2% on average by our framework.

In Figure 6.13, we present the timing performance improvements of the output kernels generated by our synchronization optimization framework compared to the original kernels running on a GTX480 GPU. The results presented in Figure 6.13 are the average timing performance improvements of these benchmarks.

As shown in Figure 6.13, the timing performance of the output kernels generated by our synchronization optimization framework is better than or the same as the original kernels or the kernels with synchronizations optimized manually. We can see that the more the redundant synchronizations be removed, the higher the timing performance improvements we achieve. For example, nearly 50% of the synchronizations executed are removed for the benchmarks, *mv*, *conv*, and *reduction*, so they achieve the highest performance improve-
ments (6.98% for mv, 5.16% for conv, and 4.27% for reduction). For the benchmarks, backprop and hotspot, only 20% of the synchronizations are removed, and the performance gains are 1.63% and 1.92% respectively. In the benchmarks, transpose and kmeans, synchronizations are located outside the loop and the influence of synchronizations on the running time is not as significant as other benchmarks though the timing performance can still be improved by removing redundant synchronizations. For the benchmarks, matmul, tmv, and imregionmax, synchronizations executed in the optimized output kernels are nearly the same as the original kernels and there are no significant timing performance improvements. To summarize, the output kernels produced by our synchronization optimization framework have the same timing performance as or better timing performance than the GPU kernel programs optimized manually.

Figure 6.14 shows that our synchronization insertion algorithm converges after 2 or 3 rounds. We can see that for most benchmark programs, our algorithm can converge after 2 iterations. Three iterations are needed for the benchmarks, mv and backprop, which are the worst cases among all the input benchmarks. This shows that our synchronization optimization framework can be used in a real compiler without degrading the performance of the compiler.
6.5 Related Work

Michael O’Boyle et al. designed the automatic barrier synchronization minimization techniques for multi-core systems that use shared memory to communicate among processor cores [49]. And barrier synchronization is used to preserve cross-core data dependencies in their compiler techniques. They proposed a very intelligent data dependency elimination and program subdivision technology that can simplify synchronization insertion. They also discussed how to apply their algorithms to the loops using a method called backward edge elimination. However, there algorithms are very complex when processing loops because different types of loops are processed in different ways. They did not provide a uniform solution for all the possible program control structures. So it is hard to implement their techniques in a real compiler. In addition, their algorithms are not targeting GPU, and cannot process the GPU kernels directly. Our proposed synchronization optimization framework can transform all the program structures of the input GPU kernels including branches and loops into SSA-like form, which makes the synchronization insertion systematic.

Yi Yang et al. [61] proposed a new source-to-source compiler framework that can apply global memory coalescing automatically. Their compiler inserts synchronization statements using the most conservative strategy which causes the insertion of redundant synchronization statements in some cases. Our proposed synchronization optimization framework automatically insert synchronization statements into GPU kernels at compile time, while minimizing the number of inserted statements.

In [65], Antonia Zhai et al. proposed a new scheme that can shorten the critical forwarding path in multi-core systems. They use CFG based analysis to determine where to properly put ‘wait’ and ‘signal’ synchronization statements. Hwansoo Han et al. also proposed a new solution to reduce the total number of synchronizations needed in a multi-thread system using fork-join model [34]. The fundamental idea of these two papers is that no synchronization is needed if there is no data communication between different cores. In our proposed compile-time synchronization optimization framework, no synchronization
is inserted when no data is shared between different threads of the input GPU kernels.

Nicolau et al. proposed technologies that could eliminate the redundant “wait” primitives in the programs for the multi-core systems in [48]. Compared to their work, we use the SSA-like form to eliminate the redundant synchronizations which could improve the efficiency of our framework.

Many techniques have been proposed for GPU synchronization optimization. In [59], a simple inter-block synchronization solution based on atomic operations in GPU systems is proposed. In [64] Yilmazer et al. proposed a new hardware solution for inter-thread block synchronization. Compared to the traditional atomic operations, the hardware solution can reduce the overhead for inter-thread block synchronization. Our synchronization optimization framework does not consider inter-thread block synchronization. In our future research, we will design a compiler framework that can insert both intra-thread block synchronizations and inter-thread block synchronization.

6.6 Summary

In this chapter, we present a synchronization optimization framework to automatically insert synchronization statements into GPU kernels at compile time, while minimizing the number of inserted synchronization statements. Experimental results show that our synchronization optimization framework achieved 100% correctness. In addition, the number of synchronization statements in the GPU kernels is reduced by 32.5%, and the number of synchronization statements executed is reduced by 28.2% on average by our framework. Our synchronization optimization framework can be integrated in a real compiler as a compiler optimization pass in automatic parallelization or in an optimization compiler framework.
Chapter 7: Compiler-assisted Software-Managed Cache Optimization Framework

7.1 Introduction

7.1.1 Motivation

As we discussed in Chapter 2, GPUs have shared memories (software-managed high-speed scratch pad memories) designed for bridging the speed gap between the GPU cores and the global off-chip memories in addition to the hardware managed caches. Each SM of a GPU has its own shared memory. GPU programmers could use the shared memory to buffer frequently used data blocks [9, 22]. The threads of the same thread block can share data among each other in the shared memory. Fully taking advantage of the shared memory can help GPU developers optimize their programs in several aspects [9, 22]:

1. Reduce the number of global memory accesses by caching a small amount of data that will be accessed repeatedly during the execution of the parallel threads.

2. Enable the threads in the same thread block to collaborate with each other with a
low-latency data transfer channel.

3. Increase the ratio of coalesced global memory accesses.

As recommended by Nvidia’s CUDA programming guide [22], the GPU programmers must try their best to replace the global memory accesses with the shared memory accesses. Compared to the hardware managed cache, buffering data in the software-managed shared memory has many advantages:

1. Data can be cached based on the needs. The GPU programmers can just bring in the memory blocks that certainly will be reused during the execution. However, the hardware cache does not distinguish between the data that will be reused and the data that will not be reused, so it wastes valuable high-speed cache resources by bringing in data blocks that will not be reused.

2. When using the shared memory, the GPU programmers can control when to discard cached memory blocks that are no longer needed, which prevents the memory blocks from being evicted before they are reused.

3. The GPU programmers can control the data layout in the shared memory, which avoids the unnecessary cache block evictions caused by the limited number of ways associated in the hardware cache.

However, as we illustrate in Section 7.1.2, using the shared memory increases the complexity of the GPU kernels and increases the workload of the GPU programmers. To fully take advantage of the shared memory, the GPU programmers must understand the data reuse patterns of their GPU applications and they must have the knowledge of the GPU hardware [62, 63, 44, 6, 57, 55]. Because the amount of the shared memory occupied by each thread block affects the total number of active CTAs on an SM [22], the programmer also needs to decide which array should be mapped to the shared memory and
calculate the amount of shared memory used by each thread block in order to achieve the best performance.

Considering the complexity of the shared memory usage, we design a compiler-assisted software-managed cache optimization framework to help the GPU programmers use the shared memory effectively. Our software-managed cache optimization framework can reduce the workload of the GPU programmers and enable them use the shared memory more efficiently. The GPU programmers can just write the GPU kernels without converting the global memory accesses to the shared memory accesses. And our compiler-assisted software-managed cache optimization framework will apply the data reuse analysis based on the polyhedron model of the GPU kernels and convert the global memory accesses to shared memory accesses automatically.

To design the compiler-assisted software-managed cache framework, several challenges need to be overcome:

1. The generalized software-managed cache framework needs to facilitate the automatic kernel optimization.

2. Parameters needed by the software-managed cache need to be calculated automatically based on the data reuse analysis of the input GPU kernel, such as the size of the shared memory buffer and the base address in the global memory of the cached data block. The analysis must be able to handle one dimensional memory accesses and multiple dimensional memory accesses.

3. The compiler framework must optimize the parameters used by the software-managed cache to achieve the best performance.

The rest of the chapter is organized as follows. Section 7.2 presents our compiler-assisted software-managed cache optimization framework. Section 7.3 illustrates how the source-to-source compiler framework calculates the parameters for the software-managed cache by using the data reuse analyzer based on the polyhedron model for the GPU kernels.
Section 7.6 presents the related work of this chapter. Section 7.4 reports our experimental results and section 7.5 presents the limitations of our current implementation. Section 7.7 concludes this chapter.

### 7.1.2 Case Study

We use a simple illustration example, the 2D convolution kernel shown in Figure 7.1, to show the complexity of the usage of the shared memory. In Figure 7.1, array $A$ is the input image to be processed and array $B$ is the convolution kernel. We define the size of the convolution kernel with two constant values $W$ and $H$. Assume the thread block size is $16 \times 16$ for illustration purpose. Using the data reuse analysis we can see that the memory accesses to array $A$ in Line 08 can be reused in later memory accesses. In addition, memory accesses to array $B$ in line 09 are not coalesced. We should convert those global memory accesses to the shared memory accesses to improve the performance as shown in Figure 7.2.

```c
#define W 32
#define H 32
__global__ void conv_original(float **A, float *B, float *C) {
    int i;
    int j;
    float sum = 0;
    for (j=0; j<H; j=j+1) {
        for (i=0; i<W; i=i+1) {
            float a;
            float b;
            a = A[idy-j+H][idx-i+W];
            b = B[W*j+i];
            sum += a*b;
        }
    }
    C(idy, idx) = sum;
}
```

Figure 7.1: A basic 2D convolution kernel
```c
#define W 32
#define H 32
__global__ void conv_opt1(float **A, float *B, float *C) {
...
01:   __shared__ bBuf[W];
02:   __shared__ aBuf[W/2][W*2];
03:   for (j=0; j < H; j = j+1) {
04:     __syncthreads();
05:     if(tid < W) {
06:       bBuf[tid] = B[j*W+tid];
07:     } 
08:     if(tidx < (W/2)){
09:       aBuf[tidy][tidx] = A[idy-j+H][bdimx*tidx+tidx];
10:      aBuf[tidy][tidx+(W/2)] = A[idy-j+H][bdimx*tidx+tidx+(W/2)];
12:      aBuf[tidy][tidx+W+(W/2)] = A[idy-j+H][bdimx*tidx+W+(W/2)];
13:    }
14:    __syncthreads();
15:    for (i=0; i < W; i = i+1) {
...
16:      a = aBuf[tidy][tidx-i+W];
17:      b = bBuf[i];
...
18:    }
19:  }
20:  C(idy, idx) = sum;
21:}
```

Figure 7.2: The optimized 2D convolution kernel without loop tiling
As illustrated in Figure 7.2, several steps are needed to convert the global memory accesses to the shared memory accesses. First, the GPU programmers must analyze the data reuse pattern of the memory locations accessed in the program to obtain the memory access footprints in the shared memory. For example, in the 2D convolution program, memory accesses to array $B$ are not coalesced. However, consecutive memory locations in array $B$ are accessed in the $i$-th loop of the program. We should put a fragment of array $B$ into the shared memory to take advantage of the coalesced memory accesses. The data reuse pattern for array $A$ is more complex. The elements in each row of array $A$ can be reused among different threads in the same thread block along the $x$ direction. In addition, each row of the array $A$ can be reused among different iterations of loop $i$. Finally, each row of array $A$ can be reused among different iterations of loop $j$ (Data reuses along the $y$ direction). The memory footprint analysis can provide us the size of the shared memory needed. As illustrated in Figure 7.2, we can cache the data needed for each iteration of loop $j$, and the sizes of the shared memory buffers are defined in Line 01 and Line 02. The optimized 2D convolution kernel shown in Figure 7.2 minimizes the shared memory usage. However, the optimized 2D convolution kernel in Figure 7.2 does not consider the data reuses along the $y$ direction.

Second, the GPU programmers must obtain the address mapping relationship between the shared memory accesses and the global memory accesses. With the mapping relationship obtained, the GPU programmers can assign dedicated threads to bring in the data need to be cached as shown in Line 04 – 14 in Figure 7.2.

Finally, the GPU programmers must convert the original global memory accesses to the shared memory accesses as shown in Line 16 and 17 in Figure 7.2.

The performance of the program in Figure 7.2 is limited because the data reuses along the $y$ direction are not considered. To further improve the performance, we introduce loop tiling to control the number of memory blocks needed to cache for multiple iterations in the $j$-th loop level as illustrated in Figure 7.3. The outer loop is tiled by a parameter “STEP”.
```c
#define W 32
#define H 32
#define STEP 4
__global__ void conv_opt1(float **A, float *B, float *C) {
    ...  
01:     __shared__ bBuf[W*STEP];
02:     __shared__ aBuf[W/2+STEP-1][W*2];
03: for(jj=0; jj < H; jj += STEP) {  
04:     __syncthreads();
05: for(j=0; j < STEP; j++) {  
06:      if(tid < W) {
07:          bBuf[j*W+tid] = B[(jj+j)*W+tid];
08:      }
09:  }
10: for(j=0; j < (W/2)+STEP+1; j++) {  
11:      if(tid<(W/2)) {  
12:          aBuf[j][tidx]=A[bidy*bdimy-(jj+j)+H][bidx*bdimx+tidx];
13:          aBuf[j][tidx+(W/2)]=A[bidy*bdimy-(jj+j)+H][bidx*bdimx+tidx+(W/2)];
15:          aBuf[j][tidx+W+(W/2)]=A[bidy*bdimy-(jj+j)+H][bidx*bdimx+tidx+W+(W/2)];
16:      }
17:  }
18:     __syncthreads();
19: for (j=0; j < STEP; j=j+1) {
20:     for (i=0; i < W; i = i+1) {
...  
21:         a = aBuf[j][tidx-i+W];
22:         b = bBuf[j*W+i];
...  
23:     }
24:  }
25: C(idy, idx) = sum;
26: }
27: }
```

Figure 7.3: The optimized 2D convolution kernel with loop tiling
Correspondingly, the statements to bring in the data to be reused into the shared memory become more complex as shown in Line 04 – 18 in Figure 7.3.

The loop tiling technology introduces another question: how to determine the best loop tiling parameter “STEP”? As we discussed in Chapter 2, the maximum number of active CTAs mapped to an SM can be determined by

$$\#CTAs = \min\left(\frac{\text{total\#registers}}{\#registersPerCTA}, \frac{\text{totalSharedMemory}}{\#sharedMemoryPerCTA}, \#hardwareCTASlots\right)$$

(7.1)

For Nvidia GPUs, the $\#hardwareCTASlots$ is equal to 8 and the size of shared memory can be configured to 16KB or 48KB for each SM [9, 22], which is shared among all of the active CTAs mapped to this SM. According to equation 7.1 [5, 22], when a thread block occupies more than 2KB or 6KB of the shared memory under these two configuration respectively, the shared memory limitation will reduce the total number of active CTAs mapped to a single SM. So the larger the shared memory buffer is, the lower performance the GPU kernel can reach. As illustrated in Figure 7.3, a basic relationship between the loop tiling parameter “STEP” and the total amount of the shared memory is derived: the greater the “STEP” is, the larger the shared memory buffer is. So the selection of “STEP” will affect the performance of the GPU kernel. When “STEP” is too large the performance will be limited by the small number of active CTAs mapped to a single SM. However, on the other hand, when “STEP” is too small, the shared memory resource is wasted because the data cached in the shared memory is not fully reused. So the best shared memory usage strategy is to use as much shared memory as possible under the condition that the number of active CTAs mapped to each SM is not reduced.

In Figure 7.4, the relationship between the shared memory buffer size and the performance of the 2D convolution benchmark is presented based on the GPU platform Nvidia Tesla K40c when the shared memory size is configured to 48KB. In Figure 7.4, “0KB”
Figure 7.4: The relationship between the buffer size and the performance

indicates the performance of the basic kernel without using the shared memory. The remaining shared buffer sizes correspond to the kernels with “STEP” set to 1, 2, 4, 8, 16 and 32 respectively as shown in Figure 7.3. The performance result supports our prediction. When the buffer size is less than 8KB, the larger the shared memory buffer is the better the performance is. The reason is that the possibility of data reuse increases as more shared memory is used. However, when the buffer size is larger than 8KB, the performance becomes worse when we increase the amount of the shared memory used. That is because the number of active CTAs is limited by the shared memory usage according to equation 7.1.

7.2 Compiler-assisted Software-managed Cache Optimization Framework

7.2.1 An Illustration Example

As illustrated in Figure 7.5(a), the buffered cache blocks in the shared memory can be represented with two parameters “BASE” and “SIZE” for 1D array in the global memory. “BASE” represent the base index of the cached blocks in the global memory and the
“SIZE” indicates the number of cache blocks in the cached buffer. To represent a cached buffer in 2D arrays, we need 4 parameters. “BASE_X” and “BASE_Y” represent the base index along the $x$ direction and the $y$ direction in the global array respectively. “SIZE_X” and “SIZE_Y” represent the number of cache blocks along the $x$ direction and the $y$ direction respectively. The representation of three or higher dimensional arrays is similar to that of the 2D arrays.

In Figure 7.6, to take advantage of the coalesced global memory accesses [9, 22], we choose 16 array elements as the basic data block unit for the software-managed cache, which is defined as the Cache_block_size. Then a data block buffered in the shared memory can be represented as a data block consisting of several consecutive cache blocks.

With the parameters defined for the software-managed cache, the compiler-assisted software-managed cache optimization framework is illustrated in Figure 7.6(b). In this software-managed cache optimization framework, loop tiling is applied, and the declaration of the shared memory buffer is generated automatically. The original kernel is presented in Figure 7.6(a). In the GPU kernel optimized by the software-managed cache optimization framework, the original loop ($i$ loop in Figure 7.6(a)) is tiled into the outer loop ($jj$ loop in Figure 7.6(b)) and the inner loop ($j$ loop in Figure 7.6(b)) to control the amount of the shared memory used. Before the kernel function body, we define the cache parameters. The
Figure 7.6: The compiler-assisted software-managed cache optimization framework
parameters “BASE_X” and “BASE_Y” represent the base array index along the $x$ direction and the $y$ direction during the execution of the inner loop in Figure 7.6(b), which can be represented by a linear function of the thread block indexes, the thread block dimension, the loop iterator of the outer loop, “STEP” and the loop boundaries of the inner loops. Because the memory footprint of each thread block is the same, the parameter “SIZE” will not be affected by the block indexes and the outer loop iterator “jj”. The parameter “STEP” will always be a constant value. And “BASE_X” and “BASE_Y” for memory accesses in array $A$ in the 2D convolution benchmark can be represented as:

\[
BASE_X = bidx \times bdimx + 1 \\
BASE_Y = (bidy \times bdimy - (jj + STEP - 1) + H)
\] (7.2)

“SIZE_X” and “SIZE_Y” can be represented as:

\[
SIZE_X = 2 \times W/\text{Cache block size} = 4 \\
SIZE_Y = bdimy + STEP - 1
\] (7.3)

In the kernel optimized by the software-managed cache optimization framework, the shared memory buffers are first defined according to the parameter “SIZE” defined before the kernel body. At the beginning of each iteration of the outer loop, the data blocks to be cached are brought into the shared memory in parallel in a coalesced manner. Then, the global memory accesses in the inner loop are converted to the shared memory accesses.

### 7.2.2 The Mapping Relationship Between the Global Memory Accesses and the Shared Memory Accesses

For 1D arrays, we assume the memory accesses in the shared memory are indexed by $local_id$ and the corresponding global memory accesses are indexed by $global_id$. During
template<class T>
_device_ T void bring_in_1d(T *A, _shared_ T * aBuf) {

01:  for(int m = tid; m < SIZE*Cache_block_size; m += numThreads){
02:    aBuf[m] = A[BASE+m];
03:  }
04:}

Figure 7.7: The bring-in function for 1D arrays

the bring-in stage, the global id can be calculated as:

\[
global\_id = BASE + local\_id
\]  \hspace{1cm} (7.4)

During the shared memory access stage, the global memory index is known to us. The corresponding shared memory index can be calculated as

\[
local\_id = global\_id - BASE
\]  \hspace{1cm} (7.5)

So we can define the bring-in function as a function template illustrated in Figure 7.7. In Figure 7.7, “tid” indicates the thread index of the threads of the current thread block, “numThreads” indicates the total number of threads in a thread block. We can see that consecutive elements in the global memory are accessed by consecutive threads, which meets the requirement of coalesced memory accesses. The original global memory access \(A[global\_id]\) can be replaced by a proxy function call \(acc\_A(aBuf, global\_id)\) which is defined as the function template illustrated in Figure 7.8.

For 2D array memory accesses, we assume the memory accesses in the shared memory are indexed by \(local\_idx\) and \(local\_idy\) and the corresponding global memory accesses are indexed by \(global\_idx\) and \(global\_idx\) and \(global\_idy\) respectively. When the data blocks to be cached are brought in the shared memory, we already knew the local indexes. The
corresponding global indexes can be calculated as

\begin{align}
\text{global}_{idx} &= BASE_{X} + local_{idx} \\
\text{global}_{idy} &= BASE_{Y} + local_{idy}
\end{align} \tag{7.6}

When accessing the cached data blocks, we can calculate the local indexes from the global indexes by

\begin{align}
local_{idx} &= global_{idx} - BASE_{X} \\
local_{idy} &= global_{idy} - BASE_{Y}
\end{align} \tag{7.7}

Thus, the bring-in function for 2D memory arrays can be defined by a function template shown in Figure 7.9. Similarly, the global memory access “A[global_{idx}][global_{idy}]” can be replaced by the proxy function call “acc_A(aBuf,global_{idx},global_{idy})”, which is defined by the function template shown in Figure 7.10.
template<class T>
__device__ T acc_A(T **aBuf, int global_idx, int global_idy) {
01: int offset_x = global_idx - BASE_X;
02: int offset_y = global_idx - BASE_Y;
03: return aBuf[offset_y][offset_x];
04:}

Figure 7.10: The accesses buffered for 2D arrays

2D Array A

New brought in rows

Rows could be reused

Evicted rows

BASE(jj)
BASE(jj+STEP)

Figure 7.11: The reuse pattern for the 2D convolution kernel

7.2.3 The Data Reuses In the software-managed Cache

We observe that, in some of the GPU kernels, the cached data blocks for different iterations overlap with each other, which provides us the opportunity to further reduce the number of global memory accesses by reusing part of the data blocks brought in the software-managed cache.

The cached data blocks of array A for two consecutive loop iterations with iteration variable “jj” and “jj+STEP” in a certain thread block of 2D convolution kernel has the reuse pattern shown in Figure 7.11. Here we assume $\text{bdim}_y = 8$ and $\text{STEP} = 1$ for illus-
Figure 7.12: The cache row mapping lookup table

As the base of the cached memory block is calculated with the equation (7.2), in the first “$jj$” iteration, memory blocks in range (1) and memory blocks in range (2) are brought in the shared memory. In the “$jj+$STEP” iteration, the memory blocks in range (2) can be reused and do not need to be brought into the shared memory again. Only memory blocks in range (3) need to be brought into the shared memory in the “$jj+$STEP” iteration. The following iterations show the similar reuse pattern.

The number of the cached rows need to be brought in after the first “$jj$” iterations can be calculated as

$$
\#\text{rows to bring in} = \text{BASE}_Yjj + \text{STEP} - \text{BASE}_Yjj = -\text{STEP} \quad (7.8)
$$

Here we use negative values to indicate the cached data range is moving upward along the Y direction. Then the number of reused rows can be calculated as

$$
\#\text{reused rows} = \text{SIZE}_Y - \text{abs}(\#\text{rows to bring in}) = bdimy - 1 \quad (7.9)
$$

Where “abs” is the function to calculate the absolute value. Equation (7.9) can be used to detect whether or not the cache blocks can be reused in the software-managed cache. If $\#\text{reused rows} > 0$, we can say that there are cache blocks that can be reused, otherwise
To reuse the cache rows without data movement, we design a cache row mapping lookup table (LUT) as illustrated in Figure 7.12 to remap the cache rows so that some of the rows can be reused. Figure 7.12 shows the mapping situation with $STEP = 1$ and $jj = 1$. Assume $bdimy = 8$ for illustration purpose. When $jj = 1$, blocks 0−6 in the last iteration ($jj = 0$) can be reused and be remapped to blocks 1−7 in the current iteration. The last block (block 7) of the last iteration is mapped to block 0 in the current iteration, which is ready to accept new data. The LUT can be stored as a one dimensional array in the shared memory. If we initialize the LUT with values from 0−7, then the remap operation can be calculated as

$$LUT[i] = (LUT[i] + \text{#rows_toBringIn}) \ MOD \ SIZE_Y$$  \hspace{1cm} (7.10)
template<class T>
_device__ T void bring_in_2d_reuse(T **A, _shared__ T * aBuf, int * LUT) {
01: if(jj == 0) {
02: for(int m = tid; m < SIZE_Y*SIZE_X*Cache_block_size; m += numThreads) {
03: int x_offset = m%(SIZE_X*Cache_block_size);
04: int y_offset = m/(SIZE_X*Cache_block_size);
05: aBuf[y_offset][x_offset] = A[BASE_X + x_offset][BASE_Y + y_offset];
06: }
07: } else {
08: if(tid < SIZE_Y) {
09: LUT[i] = (LUT[i] + #rows_to带来的) MOD SIZE_Y;
10: }
11: __syncthreads();
12: for(int m = tid; m < abs(#rows_to带来的)*SIZE_X*Cache_block_size; m += numThreads) {
13: int x_offset = m%(SIZE_X*Cache_block_size);
14: int y_offset = m/(SIZE_X*Cache_block_size);
15: y_offset = #rows_to带来的 < 0 ? y_offset : y_offset + #reused_rows;
16: aBuf[LUT[y_offset]][x_offset] = A[BASE_X + x_offset][BASE_Y + y_offset];
17: }
18: }
19: }

Figure 7.14: The bring-in function for 2D arrays with cache block reusing

Where “MOD” indicates the integer modulo operation.

The bring-in function with cache block reusing and cache row remapping in the optimized kernel is illustrated in Figure 7.14. For the first iteration, the normal bring-in operations are applied as shown in Line 01 – 06 in Figure 7.14. Our cache row mapping LUT is updated as shown in Line 08 – 11 for the remaining loop iterations. And the modified bring-in operations in Line 12 – 17 are applied just for the cache rows that are needed to be brought in. Note that at Line 16 the shared memory accesses along the y direction are remapped by the LUT.

The corresponding memory access proxy function for memory accesses of array A is shown in Figure 7.15. The cache block reuse method to take advantage of the data
7.3 Compiler Supporting The Software-Managed Cache

As we discussed in Section 7.2, converting the global memory accesses to the shared memory accesses needs to calculate multiple parameters including BASE/BASE_X/BASE_Y, SIZE/SIZE_X/SIZE_Y, and STEP. All other parameters needed can be calculated based on these parameters. In this section we present the algorithms to determine theses parameters. These parameters are computed by data reuse analysis based on the extended polyhedron model of the GPU kernels.

7.3.1 Generate BASEs

We consider the 1D arrays first. Assume a memory access is represented as “A[a]”, where “a” is the subscript of the memory access which can be represented as a linear function of

```cpp
template<class T>
_device_ T acc_Areuse(T **aBuf, int global_idx, int global_idy, int * LUT) {
  int offset_x = global_idx-BASE_X;
  int offset_y = global_idy-BASE_Y;
  return aBuf[LUT[offset_y]][offset_x];
}
```

Figure 7.15: The accesses buffered for 2D arrays with cache block reusing

reuses along the $x$ direction is similar to the method handling the data reuses along the $y$ direction. Instead of using a cache row remap LUT, a cache column remap LUT is used to facilitate the cache block reuses. The data reuses along the $x$ direction and the $y$ direction are the most representative and these data reuses are relatively easy to handle in the GPU kernels. We do not consider more complex data reuse patterns in this chapter considering the overhead that might be introduced by handling those complex data reuse patterns. In the future, we can extend the polyhedron model to handle more complex data reuse patterns.
the thread indexes and loop iterations just like the memory access subscripts represented in equation (3.7). Then BASE can be defined as the minimum possible value of variable “a”, which can be calculated as

\[
\min a = S(bidy, bidx, bdimy, bdimx, tidy, tidx, loopiterators)
\]
\[
s.t. \quad 0 \leq bidy < gdimy
\]
\[
0 \leq bidx < gdimx
\]
\[
0 \leq tidy < bdimy
\]
\[
0 \leq tidx < bdimx
\]
\[
jj \leq j < jj + STEP
\]
\[
jj \leq j < \text{END}_J
\]
\[
0 \leq jj < \text{END}_J
\]
\[
0 \leq i < \text{END}_I
\]
\[
\ldots
\]

Where gdimx, gdimy, bdimy, bdimx, bidy, bidx, STEP, jj, ENDJ, and ENDI ... are treated as parameters. Problem (7.11) is a parametric integer linear programming problem which can be solved by PipLib [29, 28].

For example, we can obtain the expression of “BASE” for the 1D memory accesses of array B in the 2D convolution kernel by solving the following integer linear programming
problem (7.11).

\[
\begin{align*}
\text{min} & \quad a = j \cdot W + i \\
\text{s.t.} & \quad 0 \leq bidy < gdimy \\
& \quad 0 \leq bidx < gdimx \\
& \quad 0 \leq tidy < bdimy \\
& \quad 0 \leq tidx < bdimx \\
& \quad jj \leq j < jj + \text{STEP} \\
& \quad jj \leq j < H \\
& \quad 0 \leq jj < H \\
& \quad 0 \leq i < W
\end{align*}
\] (7.12)

In solving the parametric linear programming problem (7.12), we can obtain the minimum value for “a” when \(j = jj\) and \(i = 0\). So the BASE can be represented as

\[\text{BASE} = W \cdot jj\] (7.13)

We can calculate the “BASE_X” and “BASE_Y” for 2D array accesses by solving the similar parametric integer linear programming problem shown in (7.11). Assume a 2D memory access is represented as “A[y][x]”, where “y” and “x” are subscripts along the \(y\) direction and the \(x\) direction respectively.

For example, “BASE_X” for memory accesses of array \(A\) in the 2D convolution kernel
can be calculated as

\[
\begin{align*}
\min & \quad x = idx - i + W = bidx * bdimx + tidx - i + W \\
\text{s.t.} & \quad 0 \leq bidy < gdimy \\
& \quad 0 \leq bidx < gdimx \\
& \quad 0 \leq tidy < bdimy \\
& \quad 0 \leq tidx < bdimx \\
& \quad jj \leq j < jj + STEP \\
& \quad jj \leq j < H \\
& \quad 0 \leq jj < H \\
& \quad 0 \leq i < W
\end{align*}
\] (7.14)

In (7.14), we can define an intermediate variable \( b = bidx * bdimx \) so that “a” is represented as a linear function since both \( bidx \) and \( bdimx \) are known parameters. We can obtain the minimum value for “a” when \( i = W - 1 \) and \( tidx = 0 \), so “BASE_X” can be represented as

\[
BASE_X = bidx * bdimx + 1
\] (7.15)

Similarly, we can calculate “BASE_Y” by
\[
\begin{align*}
\min \quad & y = idy - j + W = bidy \cdot bdimy + tidy - j + H \\
\text{s.t.} \quad & 0 \leq bidy < gdimy \\
& 0 \leq bidx < gdimx \\
& 0 \leq tidy < bdimy \\
& 0 \leq tidx < bdimx \\
& jj \leq j < jj + STEP \\
& jj \leq j < H \\
& 0 \leq jj < H \\
& 0 \leq i < W \\
\end{align*}
\] (7.16)

When \( j = jj + STEP - 1 \) and \( tidy = 0 \), we can calculate the minimum value for “\( y \)” by the following equation.

\[
BASE_Y = bidx \cdot bdimx - (jj + STEP - 1) + H
\] (7.17)

### 7.3.2 Generate SIZEs

Again we consider the 1D array first. To represent a cache block, we need to introduce two intermediate variables. \( \alpha \) represents the cache block ID and \( \beta \) represents the offset in a cache block. Then the memory access subscript “\( a \)” can be represented as

\[
a = \alpha \cdot \text{Cache\_block\_size} + \beta + BASE
\] (7.18)

Assume the first cache block has an ID of 0, then the “SIZE” can be calculated by the following parametric integer linear programming problem.
\[
\begin{align*}
    \max & \quad \alpha + 1 \\
    \text{s.t.} & \quad a = \alpha \times \text{Cache\_block\_size} + \beta + \text{BASE} \\
    & \quad a = S(bidy, bidx, bdimy, bdimx, tidy, tidx, loop\_iterators) \\
    & \quad 0 \leq bidy < gdimy \\
    & \quad 0 \leq bidx < gdimx \\
    & \quad 0 \leq tidy < bdimy \\
    & \quad 0 \leq tidx < bdimx \tag{7.19} \\
    jj & \leq j < jj + STEP \\
    jj & \leq j < END_J \\
    0 & \leq jj < END_J \\
    0 & \leq i < END_I \\
    \ldots
\end{align*}
\]

The size estimation for 2D array memory accesses along the \(x\) direction is similar to the 1D arrays.
\[ \max \alpha + 1 \]
\[ \text{s.t. } x = \alpha \times \text{Cache\_block\_size} + \beta + \text{BASE\_X} \]
\[ x = S_x(bidy, bidx, bdimy, bdimx, tidy, tidx, loopiterators) \]
\[ 0 \leq bidy < gdimy \]
\[ 0 \leq bidx < gdimx \]
\[ 0 \leq tidy < bdimy \]
\[ 0 \leq tidx < bdimx \]
\[ jj \leq j < jj + \text{STEP} \]
\[ jj \leq j < \text{END}_J \]
\[ 0 \leq jj < \text{END}_J \]
\[ 0 \leq i < \text{END}_I \]
\[ \ldots \]

The size estimation for 2D array memory accesses along the \( y \) direction is much simpler:
\[
\begin{align*}
\text{max} \quad & y - \text{BASE}_Y + 1 \\
\text{s.t.} \quad & y = S_y(bidy, bidx, bdimy, bdimx, tidy, tidx, \text{loop iterators}) \\
& 0 \leq bidy < \text{gdimy} \\
& 0 \leq bidx < \text{gdimx} \\
& 0 \leq tidy < \text{bdimy} \\
& 0 \leq tidx < \text{bdimx} \\
& jj \leq j < jj + \text{STEP} \\
& jj \leq j < \text{END}_J \\
& 0 \leq jj < \text{END}_J \\
& 0 \leq i < \text{END}_I \\
\end{align*}
\]

(7.21)

### 7.3.3 Validation Checking

A memory access is buffered by the software-managed cache only when it meets all of the following requirements:

1. The memory accesses cannot be coalesced or can be reused during the execution of the kernel. Array elements that are just accessed once should not be buffered by the shared memory.

2. The size of the shared memory buffer is less than the capacity of the shared memory.

The size check can be performed based on the size estimation discussed in Section 7.3.2. The next step is to check whether or not the memory accesses can be coalesced and to check whether or not the data blocks buffered can be reused.

**Definition 19.** Coalesced memory accesses During the execution process of a thread block, if two memory access instances \( M \) and \( M' \) of the same memory access statement meet all
of the following conditions, then the memory accesses can be coalesced.

1. \( M \) and \( M' \) are issued by two consecutive threads along the \( x \) direction.

2. \( M \) and \( M' \) are accessing consecutive array elements.

3. \( M \) and \( M' \) are issued simultaneously.

Condition 1 in Definition 19 can be represented with inequalities:

\[
\begin{align*}
0 \leq bidy, bidy' &< gdimy \\
0 \leq bidx, bidx' &< gdimx \\
0 \leq tidy, tidy' &< bdimy \\
0 \leq tidx, tidx' &< bdimx \\
bidy &= bidy' \\
bidx &= bidx' \\
tidy &= tidy' \\
tidx + 1 &= tidx'
\end{align*}
\]  

(7.22)

Condition 2 in Definition 19 can be represented with inequalities:

\[
\begin{align*}
\alpha_0 < \alpha'_0 \\
\alpha_i = \alpha'_i, \ i = 1, 2, ....
\end{align*}
\]  

(7.23)

Where \( \alpha_i \) are memory access subscripts defined in (3.6).
Condition 3 in Definition 19 can be represented with inequalities:

\[
\begin{align*}
0 \leq i_0, i'_0 &< END_0 \\
0 \leq i_1, i'_1 &< END_1 \\
\vdots &\\
i_0 &= i'_0 \\
i_1 &= i'_1 \\
\vdots &
\end{align*}
\]

(7.24)

What we are interested in is to calculate the memory accesses’ distance along the \(x\) direction. We can apply the validation check based on the solution of the following integer linear programming problem:

\[
\begin{align*}
\text{max} & \quad \alpha'_0 - \alpha_0 \\
\text{s.t.} & \quad \alpha'_0, \alpha_0 \in O
\end{align*}
\]

(7.25)

Where \(O\) is the polyhedron defined by (7.22), (7.23) and (7.24). If the solution of the integer linear programming problem 7.25 is equal to 1, then it means the maximum memory access distance along the \(x\) direction is 1 and the memory accesses can be coalesced.

**Definition 20. Data reuses in the software-managed cache** During the execution process of a thread block, if two memory access instances \(M\) and \(M'\) meet all of the following conditions, then the data cached in the shared memory will be reused

1. \(M\) and \(M'\) are issued by different threads or by the same thread at different time steps.

2. \(M\) and \(M'\) are accessing the same memory location.

The data reuses in the shared memory can be defined in Definition 20.
The first condition of Definition 20 can be represented with the following inequalities if \( M \) and \( M' \) are issued by different threads

\[
\begin{align*}
0 & \leq bidy, bidy' < gdimy \\
0 & \leq bidx, bidx' < gdimx \\
0 & \leq tidy, tidy' < bdimy \\
0 & \leq tidx, tidx' < bdimx \\
bidy & = bidy' \\
bidx & = bidx' \\
tid & = tidy \times bdimx + tidx \\
tid' & = tidy' \times bdimx' + tidx' \\
tid & < tid'
\end{align*}
\]

(7.26)

Here two intermediate variables \( tid \) and \( tid' \) are introduced to facilitate the calculation. \( tid \) and \( tid' \) represent the thread ID for \( M \) and \( M' \) respectively.

The first condition of Definition 20 can be represented with the following inequalities if \( M \) and \( M' \) are issued by the same thread at different time steps.
\[
\begin{align*}
0 &\leq bidy, bidy' < gdimy \\
0 &\leq bidx, bidx' < gdimx \\
0 &\leq tidy, tidy' < bdimy \\
0 &\leq tidx, tidx' < bdimx \\
\text{bidy} &= \text{bidy}' \\
\text{bidx} &= \text{bidx}' \\
\text{tidy} &= \text{tidy}' \\
\text{tidx} &= \text{tidx}' \\
\vec{i} &\neq \vec{i}'
\end{align*}
\]

(7.27)

The second condition of Definition 20 can be represented with the following inequalities

\[
\begin{aligned}
\{ &\alpha_i = \alpha_i', i = 0, 1, 2, \ldots \\
\}
\end{aligned}
\]

(7.28)

The data reuses in the software-managed cache can be detected by solving the following integer linear programming problem:

\[
\begin{align*}
\text{max} &\quad tid' - tid \\
\text{s.t.} &\quad tid, tid' \in R
\end{align*}
\]

(7.29)

Where \( R \) is the polyhedron defined by (7.26) or (7.26) and (7.28). If the integer linear programming problem (7.29) has a solution, then we can say that there are data reuses in the software-managed cache.
7.3.4 Obtain The Best STEP Value

As we discussed in Section 7.1, we want to use as much shared memory as possible under the condition that the usage of the shared memory does not reduce the total number of active CTAs mapped to an SM.

The total size needed for the shared memory can be calculated as

\[
\text{size} = \sum_{\text{each accesses}} \text{SIZE}_Y \times \text{SIZE}_X \times \text{Cache\_block\_size} \times \text{sizeof}(\text{element})
\] (7.30)

Here, \( \text{size} \) is a function of “STEP”, since “STEP” might influence \( \text{SIZE}_Y \) or \( \text{SIZE}_X \) or both. And we can initialize the parameter “STEP” to 1, and try to increase “STEP” by 1 each iteration, until \( \text{size} \) is greater than the shared memory limitation.

Nvidia suggests that the GPU programmers to keep at least 6 active CTAs on each SM, so when the shared memory is configured to 48KB, the shared memory limitation for each thread block is \( 48KB / 6 = 8KB \).

7.4 Evaluation

7.4.1 Experimental Platform

We implement our compiler-assisted software-managed cache optimization framework based on the GPU source-to-source compiler framework proposed by Yang et al. [61] and the polyhedron compiler platform proposed by C’dric et al. [8, 7, 13]. The polyhedron compiler platform is used to compute the software-managed cache parameters including the size of the shared memory buffer and the best loop tiling step size. The compiler-assisted software-managed cache optimization framework transforms the input GPU kernel into the optimized version based on the parameters provided by the polyhedron platform. Specifically its tasks include:
1. Generate the intermediate representation format for the polyhedron analyzer.

2. Perform loop tiling on the input GPU kernel.

3. Insert shared memory buffer declaration statements.

4. Insert code segment to bring in the data to the shared memory.

5. Insert the shared memory access code.

6. Synchronization statements are inserted by the synchronization optimization framework discussed in Chapter 6.

We select 12 benchmarks from CUDA SDK and Rodinia GPU benchmark suite [15] to evaluate the performance of our compiler-assisted software-managed cache optimization framework, which are introduced briefly as follows:

(1) conv: Two dimension convolution algorithm, whose input size is 4x4k.
(2) matmul: Block matrix multiplication kernel, whose input size is 4x4k.
(3) mv: Matrix vector multiplication kernel, whose input size is 4x4k.
(4) tmv: Transposed matrix vector multiplication kernel, whose input size is 4x4k.
(5) transpose: Matrix transpose algorithm, whose input size is 4x4k.
(6) vv: Vector vector multiplication kernel, whose input size is 16k.
(7) demosaic: Image demosaicing algorithm, whose input size is 8x8k.
(8) hotspot: A thermal simulation algorithm, whose input size is 512x512.
(9) kmeans: The kmeans cluster algorithm, whose input is 494020 records.
(10) particlefilter: The particle filter algorithm, whose input size is 100k.
(11) streamcluster: An algorithm that solves the online clustering problem, whose input size is 128k.
(12) backprop: Back propagation algorithm used in the training process of layered neural networks, whose input is 65536 points.
<table>
<thead>
<tr>
<th>module</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU frequency</td>
<td>745 MHz</td>
</tr>
<tr>
<td>CUDA parallel processing cores</td>
<td>2880</td>
</tr>
<tr>
<td>CUDA compute capability</td>
<td>3.5</td>
</tr>
<tr>
<td>GPU global memory</td>
<td>12GB</td>
</tr>
<tr>
<td>Host CPU</td>
<td>Intel i5-760</td>
</tr>
<tr>
<td>Host Memory size</td>
<td>4GB</td>
</tr>
<tr>
<td>NVCC version</td>
<td>7.0</td>
</tr>
</tbody>
</table>

Table 7.1: Hardware platform

We compare the performance of the original GPU kernels with the performance of the GPU kernels optimized by the compiler-assisted software-managed cache optimization framework. We also compare the performance of the GPU kernels optimized by the compiler-assisted software-managed cache optimization framework with the performance of the manually optimized GPU kernels and the performance of the GPU kernels optimized by Yang’s framework with shared memory used [61]. For those SDK benchmarks that are also included in Yang’s test cases (conv, matmul, mv, tmv, vv, demosaic), we choose the GPU kernels optimized by Yang’s compiler framework as the manually optimized version. For the rest of the benchmarks (hotspot, kmeans, particlefilter, streamcluster, backprop), we optimize the kernels manually to use the shared memory to cache some of the global memory accesses.

All the evaluations are performed on Nvidia’s Tesla K40 GPU system. The hardware configuration is presented in Table 7.1.

7.4.2 Experimental Results

We summarize the value of “STEP” for the test benchmarks calculated by our compiler-assisted software-managed cache optimization framework in Table 7.2, in which the corresponding software cache size are measured in bytes. Note that the relationship between the “STEP” value and the size of the software cache is not linear. That is be-
cause we always bring in a whole cache line into the shared memory to take advantage of the coalesced global memory accesses. For example, in the mv benchmark we have a 2D memory access and a 1D memory access that can be cached in the shared memory. The parameters calculated by our compiler framework are $SIZE_A_X = ((STEP - 1)/COALESCE\_WIDTH + 1)$, $SIZE_A_Y = bdimy$ and $SIZE_B = ((STEP - 1)/COALESCE\_WIDTH + 1)$. Assume $bdimy = 16$, and the coalescing width $COALESCE\_WIDTH = 16$ for illustration purpose, then the total cache size consumed is 2176 bytes when $STEP = 15$ or $STEP = 16$. However, the cache size consumed is 3264 bytes when $STEP = 17$, which is greater than the shared memory size $16k/6 = 2730\, \text{bytes}$ when the shared memory is configured to $16k$.

From Table 7.2, we can see that the mechanism to obtain the best STEP size illustrated in Section 7.3.4 works as expected. It tries to use as much shared memory as possible under the shared memory limitation we have defined. The benchmarks conv, matmul, mv, tmv, particlefilter and streamcluster have many loop iterations on the first loop level to tune the value of “STEP”, and the amount of the shared memory used is close to the software cache size limitations ($2730\, \text{bytes}$ for $16k$ shared memory configuration and $8192\, \text{bytes}$ for $48k$ shared memory configuration). For the rest of the benchmarks, the amount of the shared

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>STEP (16k)</th>
<th>STEP (48k)</th>
<th>cache size (16k)</th>
<th>cache size (48k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv</td>
<td>1</td>
<td>9</td>
<td>3264</td>
<td>5856</td>
</tr>
<tr>
<td>matmul</td>
<td>16</td>
<td>64</td>
<td>2048</td>
<td>8192</td>
</tr>
<tr>
<td>mv</td>
<td>16</td>
<td>56</td>
<td>2176</td>
<td>7616</td>
</tr>
<tr>
<td>tmv</td>
<td>512</td>
<td>2048</td>
<td>2048</td>
<td>8192</td>
</tr>
<tr>
<td>transpose</td>
<td>1</td>
<td>1</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>vv</td>
<td>1</td>
<td>1</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>demosaic</td>
<td>3</td>
<td>3</td>
<td>2508</td>
<td>2508</td>
</tr>
<tr>
<td>hotspot</td>
<td>1</td>
<td>1</td>
<td>2320</td>
<td>2320</td>
</tr>
<tr>
<td>kmeans</td>
<td>1</td>
<td>1</td>
<td>192</td>
<td>192</td>
</tr>
<tr>
<td>particlefilter</td>
<td>256</td>
<td>1024</td>
<td>2048</td>
<td>8192</td>
</tr>
<tr>
<td>streamcluster</td>
<td>512</td>
<td>2048</td>
<td>2048</td>
<td>8192</td>
</tr>
<tr>
<td>backprop</td>
<td>1</td>
<td>1</td>
<td>192</td>
<td>192</td>
</tr>
</tbody>
</table>

Table 7.2: STEP value and cache size for the 16k and 48k software cache configuration
memory used is below the software cache size limitation since we do not have enough iterations to tune the value of “STEP”. Our compiler framework still tries to use as much shared memory as possible.

We evaluate the performance by measuring the execution time (which did not include the data transfer time) of those selected kernels by CUDA event functions. We test our compiler-assisted software-managed cache optimization framework with the shared memory size configured to 16k and 48k respectively. All the performance improvements are reported as speedups compared to the original GPU kernels.

We compare the speedups of the GPU kernels optimized by our compiler-assisted software-managed cache optimization framework to the manually optimized GPU kernels in Figure 7.16 with the shared memory configured to 16k and in Figure 7.17 with the shared memory configured to 48k. In Figure 7.16 and Figure 7.17, the second column shows the normalized speedups of manually optimized kernels compared to the input GPU kernels and the third column shows the normalized speedups of the GPU kernels optimized by our compiler-assisted software-managed cache optimization framework.

The shared memory usage will affect the performance of the GPU kernels in multiple aspects, and it can improve the overall performance or degrade the overall performance of
the GPU kernels. First, it can improve the global memory access efficiency by coalescing global memory accesses. Second, the data buffered in the shared memory can be reused during the execution of the GPU kernels and reduce the total number of global memory accesses. Third, too much shared memory usage might reduce the total number of active CTAs on each SM, which will reduce the overall performance of the GPU kernels. However, too less shared memory usage can also reduce the overall performance by not fully taking advantage of the shared memory resources. Finally, the shared memory usage introduces additional statements such as memory block copies and synchronizations.

Yang’s compiler framework considered both coalescing issue and data reuse issue, but it did not consider the number of active CTAs mapped to an SM. Our compiler-assisted software-managed cache optimization framework considered both memory coalescing and data reuses in the shared memory. In addition, our compiler framework can select the best loop tiling step size based on the total software cache size to control the usage of shared memory to avoid reducing the number of CTAs mapped to an SM. However, Yang’s compiler framework can take advantage of the data reuses in the shared memory by the kernel fusion mechanism for some of the GPU kernels. The kernel fusion mechanism can resize the GPU kernel block to take advantage of the data reuses along a certain direction. Our compiler-assisted software-managed cache optimization framework cannot resize the GPU
kernel blocks. Compared to the manually optimized GPU kernels, our compiler framework might introduce extra memory address mapping statements and might bring in some extra data blocks to the shared memory for some of the GPU kernels, which might affect the overall performance.

As illustrated in Figure 7.16 and Figure 7.17, the average speedup of the GPU kernels optimized by our compiler-assisted software-managed cache optimization framework over the original GPU kernels is 2.01x when the shared memory is configured to 16\(k\) and 2.03x when the shared memory is configured to 48\(k\) respectively. The average speedup of manually optimized kernels over the input GPU kernels is 2.52x when the shared memory is configured to 16\(k\) and 2.54x when the shared memory is configured to 48\(k\). The maximum speedup of the GPU kernels optimized by our compiler-assisted software-managed cache optimization framework is 6.04x for both 16\(k\) and 48\(k\) shared memory configuration. And the maximum speedup of the manually optimized GPU kernels is 5.58x and 6.04x for 16\(k\) and 48\(k\) shared memory configuration respectively. We can see that compared to the original GPU kernels, the software-managed cache optimization framework can improve the overall performance significantly by taking advantage of the shared memory. Compared to manually optimized GPU kernels or the GPU kernels optimized by Yang’s compiler framework, our compiler framework can also obtain comparable performance.

According to Figure 7.16 and Figure 7.17, our compiler-assisted software-managed cache optimization framework achieves better performance improvements for benchmarks “conv”, “matmul”, “mv”, “transpose”, “kmeans”, “particlefilter” and “bp”. The average speedup for these benchmarks is 2.66x for 16\(k\) shared memory configuration and 2.71x for 48\(K\) shared memory configuration, which is much higher than the rest of the benchmarks (1.08x for 16\(k\) configuration and 1.09x for 48\(k\) configuration on average). The common features of these benchmarks is the global memory accesses in the original GPU kernel are not coalesced. Our compiler-assisted software-managed cache optimization framework can take advantage of both coalesced global memory accesses and data reuses in the shared
memory. However, for the rest of the benchmarks, the global memory accesses are already coalesced in the original GPU kernels. Our compiler-assisted software-managed cache optimization framework can only improve the overall GPU kernel performance by reusing data in the shared memory.

The overall performance improvement for 48k shared memory configuration is higher than the 16k shared memory configuration, since the compiler-assisted software-managed cache optimization framework can adjust the amount of shared memory used automatically, for example, the benchmarks “matmul” (2.09x for 16k shared memory configuration, 2.29x for 48k shared memory configuration), “mv” (6.03x for 16k shared memory configuration, 6.04x for 48k shared memory configuration), “conv” (3.48x for 16k shared memory configuration, 3.50x for 48k shared memory configuration) and “particlefilter” (1.86x for 16k shared memory configuration, 1.92x for 48k shared memory configuration). That is because the size of the shared memory buffer affects the data reuse degree in those benchmarks and the number of loops in a GPU kernel is not limited. In the rest of the benchmarks, there are no loops in the GPU kernels (“transpose”, “vv”, “hotspot”, “kmeans”, “backprop”) or the number of loop iterations is limited (“demosaic”), which limits the amount of the shared memory used. So the performance gain with extra shared memory space is not so significant compared to the rest of the benchmarks.

7.5 Limitations

The experimental results verify our prediction that polyhedron models can be used to analyze the data reuse patterns in the GPU kernels and guide the automatic software-managed cache optimization. However, several problems limit further performance improvements:

1. Our compiler-assisted software-managed cache optimization framework can only apply loop tiling for the outermost loop level in the GPU kernels, which limits the ability to further adjust the amount of the shared memory used.
2. We have not implemented the thread block reshaping in our compiler-assisted software-managed cache optimization framework yet, which is adopted by Yang’s framework [62, 63, 61]. The GPU thread block reshaping could be used to further improve the data reuses in the shared memory. With the support of the polyhedron model analysis, the GPU thread block reshaping could be applied automatically if more precise data reuse patterns could be obtained.

3. Compared to the manual optimization, more detailed software cache footprint analysis is needed to handle the situation where the buffered memory block is not aligned to the software cache block boundaries, which could be used to reduce the chance to bring in accessory values.

### 7.6 Related Work

Yang et al. [61, 63] proposed a new source-to-source optimization compiler framework for GPUs, in which several GPU kernel optimization techniques are combined to enhance data reuses and improve the overall performance of the input GPU kernels, such as coalesced memory accesses, thread block merge or reshape, shared memory partition camping elimination and loop unrolling. Their compiler framework can improve the overall performance for many selected benchmarks. However, the program transformations in their compiler framework is performed under an experience based pattern recognition mechanism that can only handle certain types of memory accesses. Compared to Yang’s framework, we use polyhedron baesd data reuse analysis to guide the software-managed cache optimization framework that can handle all the programs with affine memory accesses. In addition, their compiler framework does not tune the best shared memory buffer size to obtain the best performance and their framework does not consider the partial data block replacement in the shared memory.

Many automatic parallelization compiler frameworks [44, 6, 57] have been proposed
to transform serial loops into parallel GPU threads, which also includes memory access mappings based on the GPU memory hierarchy. The Par4all compiler framework proposed by Amini et al. [44] mainly focuses on parallel thread mapping from serial code to parallel code. They do not consider to exploit the coalesced global memory accesses and the data reuses in the shared memory to improve the memory access performance. All the memory accesses in Par4all are performed in the global memory and the data reuses in the hardware cache is the only way to improve memory access performance, which degrades the overall performance of their optimized GPU programs.

The C-to-CUDA compiler framework proposed by Baskaran et al. [6] aims to take advantage of the shared memory in the GPU systems. However, they use a simple strategy that maps every array access in the original program to the shared memory, which might cause excessive amount of shared memory usage that leads to invalid GPU codes or low performance GPU codes with a few active CTAs working on an SM. In addition, buffering coalesced global memory accesses that do not exhibit any spatial or temporal data reuses cannot improve the overall performance, instead it might degrade the overall performance, since buffering data blocks in the shared memory introduces extra overhead [57].

The PPCG compiler framework proposed by Sven et al. [57] adopts a more sophisticated strategy to map global memory accesses to the shared memory accesses. Only uncoalesced global memory accesses with certain type of data reuses in the shared memory will be mapped to the shared memory. However, compared to our framework and Yang’s framework, PPCG compiler framework can only perform simple data reuse analysis and cache buffer size estimation to keep the generated codes as simple as possible. PPCG compiler framework does not consider buffer size tuning and partial cache block replacement to further exploit the data reuses.

In [55], Mark et al. presents a software-managed cache technique that is specially designed for sum-product algorithm on the GPU platforms, in which a tag-based cache block replacement strategy similar to the hardware cache is designed for the software-managed
cache in the shared memory. This method is suitable to the sum-product algorithm, in which most of the memory accesses are randomly distributed within a certain spatial range. Although this strategy can mostly enhance flexibility of the data reuse patterns, their technique cannot be generalized for other types of GPU programs such as 2d convolution, because it introduces a large processing overhead to manage the tag-based cache blocks. Compared to their mechanism, our polyhedron based technique can also exhibit certain level of flexibility that make our compiler-assisted software-managed cache optimization framework adapt to the data reuse patterns of the input GPU programs and keep the software cache management codes as simple as possible simultaneously.

7.7 Summary

In this chapter, we describe a compiler-assisted software-managed cache optimization framework that can take advantage of the shared memory of the GPUs automatically. Based on the polyhedron model analysis, our compiler framework can identify the global memory accesses that can take advantage of the shared memory buffers, and adjust the shared memory parameters and transform the global memory accesses to the shared memory accesses automatically. The experimental results show that our compiler-assisted software-managed cache optimization framework can improve the overall performance of the input GPU programs by $2.01x$ on average. The GPU kernels optimized by our compiler-assisted software-managed cache optimization framework obtain comparable performance improvements with the GPU kernels optimized by Yang’s compiler framework [61, 63] or manually optimized GPU kernels (the speedup is $2.52x$ on average).
Chapter 8: Conclusions and Future Work

8.1 Conclusions

In this dissertation, a Compiler Optimization Framework based on Polyhedron Model for GPGPUs is developed to bridge the speed gap between the GPU cores and the off-chip memory to improve the overall performance of the GPU systems. We extend the polyhedron model for CPU programs to the polyhedron model for GPU programs. The extended polyhedron model for GPU programs is used to detect intra-warp data dependencies, inter-warp data dependencies, inter thread block data dependencies, and to do data reuse analysis. The data reuse analyzers are used in the compiler framework to guide the automatic GPU program optimization. The optimization compiler framework includes a detailed data reuse analyzer based on the extended polyhedron model for GPU kernels, a compiler-assisted programmable warp scheduler, a compiler-assisted CTA mapping scheme, a compiler-assisted software-managed cache optimization framework, and a compiler-assisted automatic synchronization optimization framework to help the GPU programmers optimize their GPU kernels.

In Chapter 4, a compiler-assisted programmable warp scheduler framework is designed to increase the data reuses in the L1 cache of GPGPUs. Intra-warp and inter-war
data reuses are analyzed, and the parallel polyhedron model for GPU programs is also used to estimate the L1 cache reuse footprint to determine the best parameter for the compiler-assisted programmable warp scheduler, so the programmable warp scheduler can take advantage of these intra-warp and inter-warp data reuses simultaneously to improve the system performance.

In Chapter 5, to extend the programmable warp scheduler which can only take advantage of the data reuses inside the same thread block, a compiler-assisted CTA mapping framework is designed to improve the data locality among different CTAs mapped to the same SM. An inter-CTA data reuse analyzer is designed to detect the data reuse patterns of the input GPU kernels.

In Chapter 6, a compiler-assisted synchronization optimization framework is designed to insert barrier synchronizations into the GPU kernels automatically, while minimizing the total number of barrier synchronizations simultaneously. The parallel polyhedron model is used to detect cross warp data dependencies which do need barrier synchronizations to preserve the data dependencies.

Finally, in Chapter 7, a compiler-assisted software-managed cache optimization framework is designed to take advantage of the high-speed private shared memory of each SM in the GPUs, which can improve the GPU kernel performance automatically and reduce the workload of GPU programmers. A detailed data reuse analyzer is designed based on the parallel polyhedron model to analyze the data reuse footprint during the execution of GPU programs and provide the compiler framework the needed parameters to guide the shared memory usage.

The optimization compiler framework is implemented based on the extended CETUS environment [36] and evaluated on the GPU simulator, GPGPU-sim, or on a real GPU platform. Experimental results show that our optimization compiler framework can automatically optimize the GPU kernel programs and correspondingly improve the performance of the GPU systems significantly.
8.2 Future Work

In the future, we would add the automatic parallelization component into our optimization compiler framework so that the C/C++ programs could be automatically converted to the GPU kernel programs. We would further improve our optimization compiler framework to apply automatic performance tuning based on the given GPU hardware architecture and automatically apply the best program optimization for a given input GPU kernel.

Second, to simplify the hardware cache block reuse analysis, we currently do not consider the constraints posted by the “tag” section of memory addresses, since the memory locations accessed in one thread block usually are within a certain address range, i.e., the tag section of these memory addresses would be the same. The simplified analysis model is acceptable because in general we could assume the footprint of reused data blocks do not exceed the cache boundary during a relatively short time period that those data could be reused. However, this assumption still affects the accuracy of hardware cache reuse analysis. In the future we will extend the current polyhedron model to consider the constraints posted by the “tag” section of the memory addresses, which would further improve the accuracy of our hardware cache data reuse analysis.

Third, we will try to find solutions to overcome the limitations of the compiler-assisted software-managed cache optimization framework illustrated in Section 7.5. We will extend the polyhedron model to perform data reuse analysis for the kernels with multiple-level loop tiling. We will also explore the GPU kernel reshaping guided by the parallel polyhedron model.

Finally, GPU architectures are evolving quickly. The optimization compiler framework presented in this dissertation could be used as a base compiler framework, and extended as necessary to take advantage of the newly designed GPU architectures.
Bibliography


on Principles of Programming Languages, POPL ’77, pages 72–85, New York, NY, USA, 1977. ACM.


[59] Shucai Xiao and Wu chun Feng. Inter-block gpu communication via fast barrier synchronization.


