ABSTRACT

YANG, YI. Architectural Support and Compiler Optimization for Many-Core Architectures. (Under the direction of Dr. Huiyang Zhou).

Many-core architectures, such as general purpose computation on graphics processing units (GPGPU) and Intel Many Integrated Core (MIC), have been exploited to achieve teraflops computation capability on a single chip. This dissertation proposes both architectural improvement and compiler optimization for many-core architectures.

First, in order to fully utilize the power of GPGPUs, application developers have to consider the platform-specific optimization very carefully. To relieve the workload from application developer, we develop a source to source compiler, which takes a fine-grain GPGPU program as the input and generates an optimized GPGPU program by applying a set of optimization techniques.

Secondly, Intel MIC employs directive-based programming model, aiming at simplifying the program development. However when adapting the legacy programs to Intel MIC, several issues need to be addressed: 1) how to identify the profitable and parallelizable code sections for Intel MIC; 2) how to automatically generate the MIC program; 3) how to minimizing the memory transfer between the CPU and the MIC. We develop one compiler framework, called Apricot, to facilitate the program development by addressing these issues.

Thirdly, shared memory is a software-managed cache of GPGPUs and critical to the performance of GPGPU program. We advocate three software solutions and one hardware solution to mitigate the impact of poor thread level parallelism (TLP) caused by heavy usage of shared memory. While our software approaches work on existing GPGPU hardware, our hardware approach shows significant performance benefit with small hardware cost.
Last, we model the fused heterogeneous architecture by integrating a CPU and a GPU into a single chip with shared last level cache and off-chip memory. Then we advocate using the idle CPU to prefetch data into the last level cache for GPGPU programs. The experimental results show that our proposed technique can greatly improve the GPU programs.
Architectural Support and Compiler Optimization for Many-Core Architectures

by
Yi Yang

A dissertation submitted to the Graduate Faculty of North Carolina State University in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina
2013

APPROVED BY:

Huiyang Zhou
Committee Chair

Rainer Mueller

Yan Solihin

James Tuck
DEDICATION

To my daughter, Shirley Yang, my wife, Fang Ye, and my parents.
BIOGRAPHY

Yi Yang was born in Shanggao, Jiangxi, P.R.China. He received his Bachelor of Computer Science degree from the Special Class for the Gifted Young of University of Science and Technology of China in 2002 and Master of Engineering degree from Institute Of Computing Technology, Chinese Academy of Sciences in 2005. After working in China for three years, he joined University of Central Florida as a Ph.D. student in 2008 and transferred to North Carolina State University in 2009 with his advisor Dr. Huiyang Zhou. In 2013 spring, he completed his Ph.D. degree in Computer Engineering and joined Department of Computing Systems Architecture of NEC Labs as a research staff of member.
ACKNOWLEDGMENTS

I would like to thank my Ph.D. advisor, Dr. Huiyang Zhou, who has been the most important people for my graduate research. I want to show my appreciation to Dr. Huiyang Zhou, Dr. Frank Mueller, Dr. Yan Solihin, and Dr. James Tuck for serving on my dissertation committee. I gratefully thank Mike Mantor, Norman Rubin, Chris Reeve, Nishkam Ravi, Srimat Chakradhar, Qunfeng Dong for their valuable advice. In addition, I would like to thank my M.S. advisor Dr. Xiaofang Zhao, who helped me build the background knowledge for my Ph.D. study.

Many thanks go to my colleagues including Ping Xiang, Saurabh Gupta, Zhaoshi Zheng, Jingfei Kong, Martin Dimitrov and Lei Wu for their helps during my Ph.D. study.

I would like to thank my daughter, Shirley Yang, my wife, Fang Ye, my father, Yunqian Yang and my mother, Zuoping Liao for always believing in me and supporting me.
# TABLE OF CONTENTS

**LIST OF TABLES** .............................................................................................................................................................. viii

**LIST OF FIGURES** .................................................................................................................................................................... ix

**Chapter 1 Introduction**........................................................................................................................................................... 1
  1.1 Introduction........................................................................................................................................................................ 1
  1.2 Contributions ................................................................................................................................................................. 3
  1.3 Outlines........................................................................................................................................................................... 4

**Chapter 2 Background** ........................................................................................................................................................ 5
  2.1 GPGPU Architecture ......................................................................................................................................................... 5
  2.2 Intel MIC ............................................................................................................................................................................ 10
  2.3 CPU-GPU Fused Architecture ........................................................................................................................................ 11
  2.4 Related Work ................................................................................................................................................................. 11

**Chapter 3 A GPGPU Compiler for Memory Optimization and Parallelism Management** ........................................................................ 13
  3.1 Introduction........................................................................................................................................................................ 13
  3.2 Background...................................................................................................................................................................... 15
  3.3 An Optimizing GPGPU Compiler .................................................................................................................................. 16
    3.3.1 Vectorization of Memory Accesses ........................................................................................................................ 18
    3.3.2 Checking Memory Coalescing .................................................................................................................................. 19
    3.3.3 Converting Non-Coalesced Accesses into Coalesced Ones ...................................................................................... 21
    3.3.4 Data Dependencies and Data Sharing ..................................................................................................................... 23
    3.3.5 Thread/Thread-Block Merge to Enhance Memory Reuse .......................................................................................... 25
      3.3.5.1 Thread-block merge .............................................................................................................................................. 26
      3.3.5.2 Thread merge ....................................................................................................................................................... 26
      3.3.5.3 Selection between thread merge and thread-block merge .................................................................................. 28
    3.3.6 Data Prefetching ......................................................................................................................................................... 28
    3.3.7 Eliminating Partition Camping .................................................................................................................................. 29
  3.4 Design Space Exploration ................................................................................................................................................ 31
    3.4.1 The Number of Threads in a Thread Block ................................................................................................................ 31
    3.4.2 Hardware Specification ................................................................................................................................................ 32
  3.5 Case Study: Matrix Multiplication ...................................................................................................................................... 32
  3.6 Experiments ....................................................................................................................................................................... 34
    3.6.1 Experimental Methodology ........................................................................................................................................ 34
    3.6.2 Experimental Results .................................................................................................................................................. 35
  3.7 Limitations ......................................................................................................................................................................... 39
  3.8 Related Work .................................................................................................................................................................... 40
  3.9 Conclusions ....................................................................................................................................................................... 41

**Chapter 4 Apricot: An Optimizing Compiler and Productivity Tool for x86-compatible Many-core Coprocessors** ........................................................................ 42
  4.1 Introduction....................................................................................................................................................................... 42
  4.2 Background....................................................................................................................................................................... 45
Chapter 5 Shared Memory Multiplexing: A Novel Way to Improve GPGPU Performance

5.1 Introduction .................................................................................................................. 67
5.2 Background .................................................................................................................. 68
5.3 Characterization of Shared Memory Usage ................................................................... 69
5.4 Shared Memory Multiplexing: Software approaches .................................................. 72
   5.4.1 Virtual Thread Block (VTB) .................................................................................. 73
   5.4.2 Pipelined Virtual Thread Block (VTB_PIPE) ......................................................... 76
   5.4.3 Collaborative Virtual Thread Block (CO_VTB) ....................................................... 77
5.5 Shared memory Multiplexing: A Hardware Solution .................................................. 80
5.6 Experimental Methodology ......................................................................................... 84
5.7 Experimental Results .................................................................................................. 86
   5.7.1 Evaluation of Software-Based Shared Memory Multiplexing ................................. 86
   5.7.2 Evaluation of Hardware-Supported Shared Memory Multiplexing ......................... 88
5.8 Related Work ............................................................................................................... 90
5.9 Conclusion ................................................................................................................... 90

Chapter 6 CPU-Assisted GPGPU on Fused CPU-GPU Architecture

6.1 Introduction .................................................................................................................. 92
6.2 Background .................................................................................................................. 94
   6.2.1 Fused CPU-GPU architectures ............................................................................. 94
6.3 Architectural Modeling and Experimental Methodology ............................................. 95
6.4 CPU-Assisted GPGPU ................................................................................................. 97
   6.4.1 Generating the CPU Pre-Execution Code from LWST GPU Kernels ....................... 97
   6.4.2 Mechanisms to Control How Far the CPU Code Can Run Ahead of GPU Threads ... 101
   6.4.3 Generating the CPU Pre-Execution Code from HWST GPU Kernels ..................... 104
6.5 Experimental Results ................................................................................................. 108
6.5.1 Performance of CPU-Assisted GPGPU ................................................................. 108
6.5.2 The Efficacy of Data Prefetching using CPU Pre-execution .............................. 109
6.5.3 Understanding the Impact of GPU Architectures .............................................. 111
6.5.4 Sensitivity of the Parameters in CPU Pre-Execution Program ....................... 114
6.5.5 Using CPU to Execute GPU Threads ................................................................. 115
6.6 Related work ......................................................................................................... 115
6.7 Chapter Conclusion .............................................................................................. 116

Chapter 7 Conclusion ............................................................................................... 118

REFERENCES ............................................................................................................. 120
LIST OF TABLES

Table 3.1. A list of the algorithms optimized with our compiler................................. 34
Table 4.1. Main LEO keywords– Programming Language Extensions for MIC .............. 46
Table 5.1. Benchmarks for shared memory multiplexing............................................. 70
Table 6.1. Benchmarks for CPU-assisted GPGPU ..................................................... 96
LIST OF FIGURES

Figure 2.1. Overview of GPGPU architecture ................................................................. 5
Figure 2.2. The detailed structure of each SM .............................................................. 7
Figure 2.3. The performances of GPU programs after removing the share memory usage .... 8
Figure 2.4. CUDA code example .................................................................................. 9
Figure 2.5 Code example for Intel MIC ........................................................................ 10
Figure 5.1. The portion of the execution time, during which the allocated shared memory is actually utilized, of a thread block for GPGPU applications .............................................. 71
Figure 5.2. The pseudo code of a 1k-point FFT implementation, which uses 8736-Byte shared memory per thread block and there are 64 threads per thread block ...................... 72
Figure 5.3. The pseudo code of a 1k-point FFT implementation using VTB. Each thread block uses 8736-Byte shared memory and there are 128 threads in each thread block ....... 73
Figure 5.4. A comparison of execution time of the baseline to our proposed software approaches: (a) the baseline, (b) VTB, (c) VTB_pipe, and (d) CO-VTB .............................................. 75
Figure 5.5. The pseudo code of a 1k-point FFT implementation using VTB_pipe. Each thread block uses 8736-Byte shared memory and there are 128 threads in each thread block .................................. 77
Figure 5.6. The simplified pseudo code of MarchingCubes, (a) the baseline kernel code; (b) the code after we apply CO-VTB ...................................................................................... 79
Figure 5.7. A shared memory management (SMM) table. ............................................. 81
Figure 5.8. The pseudo code for 1k-point FFT kernel using the hardware supported dynamic shared memory management. Each TB allocates and de-allocates 8376-byte shared memory and contains 64 threads ........................................................................ 81
Figure 5.9. Two TBs running concurrently on an SM using dynamic shared memory management ................................................................................................................. 82
Figure 5.10. The SMM table manages shared memory usage for collaborative TBs. The first three entries are for the private shared memory of the 3 TBs. The last two entries are for the public shared memory .............................................................................. 83
Figure 5.11. Speedups of the proposed software approaches over the baseline for different GPUs ....................................................................................................................... 85

Figure 5.12. Performance comparison of MV among the baseline (xK_BL), VTB_pipe (xK_VTB_pipe) and CUBLAS 4.0 on GTX 480. ‘xK’ denotes the size of shared memory .................................................................................................................................. 87

Figure 5.13. Performance comparison of FFT among the baseline (xK_BL), VTB_pipe (xK_VTB_pipe) and CUFFT 4.0 on GTX 480. ‘xK’ denotes the size of shared memory..... 88

Figure 5.14. Speedups of hardware-supported dynamic shared memory management over baseline. .......................................................................................................................... 89

Figure 5.15. The impact of the maximum number of TBs to be allowed to run in an SM..... 90

Figure 6.1: A fused CPU-GPU architecture with a shared on-chip L3 cache and off-chip memory .................................................................................................................................. 94

Figure 6.2. The compiler algorithm to generate CPU pre-execution program from LWST GPU kernels. ........................................................................................................................... 98

Figure 6.3. A code example for LWST. (a) A vector-add GPU kernel; (b) the (partial) CPU pre-execution program. .................................................................................................. 99

Figure 6.4. Comparing memory requests generated from different versions of CPU code and from the GPU for every 100,000 cycles. “un_N” means the CPU pre-execution program with the unroll factor of N; “un_N_pref” means the CPU pre-execution program with the unroll factor of N and with the CPU L2 cache next-line prefetch enabled. ................................................................................................................................. 101

Figure 6.5. Adaptive update of the variable ‘skip_factor’ ............................................. 102

Figure 6.6. The value of the variable ‘skip_factor’ over time using our adaptive update approach shown in Figure 6.5 ................................................................................................................................. 103

Figure 6.7. The CPU pre-execution code for the vector-add GPU kernel with a fixed skip factor of 160................................................................................................................................. 104

Figure 6.8. The compiler algorithm to generate CPU pre-execution program from HWST GPU kernels ................................................................................................................................. 105
Figure 6.9. A code example for HWST. (a) A Transpose Matrix Vector Multiplication GPU kernel; (b) the (partial) CPU pre-execution program.............................. 107
Figure 6.10. GPU performance comparison among no-pre execution (no-preex), CPU pre execution with adaptive update of ‘skip_factor’ (adaptive) and CPU pre execution with a fixed ‘skip_factor’ determined from profiling (profiling) ......................... 108
Figure 6.11. L3 hit rate for GPU execution without execution (no-preex) and with CPU pre execution (adaptive)................................................................................. 110
Figure 6.12. Prefetch accuracy of CPU pre-execution. ........................................ 110
Figure 6.13. The ratio of (number of instruction executed by CPU / number of instruction executed by GPU). ........................................................................................................... 111
Figure 6.14. The speedups from CPU pre-execution for GPUs running at different frequencies and the normalized execution time without pre-execution........................................ 112
Figure 6.15. The speedups from CPU pre-execution for off-chip memory running at different frequencies and the normalized execution time without pre-execution.......... 113
Figure 6.16. The speedups from CPU pre-execution for different numbers of SPs in an SM...................................................................................................................... 114
Figure 6.17. The GPU performance for CPU pre-execution using different batch_sizes. ... 114
Figure 6.18. The speedups of workload distribution between GPU and CPU over GPU-only execution.............................................................................................................. 115
Chapter 1

Introduction

1.1 Introduction

Many-core architectures have been exploited to reach teraflops computation capability in a single chip. Such examples including NVIDIA and AMD general purpose computation on graphics processor units (GPGPUs), and Intel Xeon Phi [33], all employ in-order pipeline and contain hundreds of ALU units in one chip running at lower frequency compared with central processing units (CPUs). In order to take the advantage of the large number of ALU units, single program multiple data (SPMD) languages, such as Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL), are proposed. Recently directive based languages including OpenACC and Intel Language Extensions for Offload (LEO), are targeted at further simplifying the program development. In this dissertation, we revisit the architecture of many-core processes, identify the performance bottleneck of many-core programs and then develop compile solutions as well as the architectural improvements to address these issues so as to fully utilize the hardware platforms, primarily focusing on GPGPUs. We also discuss the challenges and opportunities on the heterogeneous architectures, on which CPUs and GPUs are integrated into same chip.

First, CUDA [51] and OpenCL [58] have been successful in making massive computational elements of GPGPUs accessible to general purpose computation. However, the application developers need to not only write the program, which is functionally correct, but also optimize the program by applying platform-specific techniques. We first identify these issues by studying open source projects and manually improving GPGPU programs [39] [86]. Then in order to relieve the job from the application developers, we propose our compiler frameworks [82] [83] which consists of a set of optimization techniques. The input of our compiler is a naïve GPU program working correctly without any optimization. Data unit vectorization and memory coalescing are applied to the naïve GPU program in order to improve the off-chip memory bandwidth. Then the compiler analyses the data sharing among the threads or across thread blocks. Based on the data sharing information, the compiler improves the data reuse using either the on-chip registers or shared memory by merging threads or
thread blocks. In addition, the compiler also supports data prefetching to overlap the long off-chip memory latency. Last, the compiler applies partition camping elimination to further improve the off-chip memory bandwidth. The experiment results on NVIDIA GPUs show that the GPU programs generated by our compiler can achieve up to 128 times speedup over naïve version, and even outperform the fine-tuned library. The compiler is further extended to support OpenCL on AMD GPUs [84].

Secondly, similar to GPGPUs, Intel Many Integrated Core (MIC) is attached to the CPU through PCIE slot. Since the memory bandwidth of PCIE is far less than the memory bandwidth inside of Intel MIC, the memory transfer between CPU and MIC can be critical to the performances of MIC programs. We develop one source to source compiler framework, called Apricot [62], to optimize memory transfer between CPU and MIC, therefore improving the performances of MIC programs. To further reduce the workload of application developers, our compiler can also identify profitable code sections using a cost model and automatically insert Language Extensions for Offload (LEO) clauses for these parallel code sections. We evaluate the benchmarks from SpecOMP and NAS parallel suits, and the experiment results show up to 24 times speedup on first MIC prototype Knights Ferry.

Thirdly, GPGPU programs utilize shared memory as one software-managed cache to either provide the communication between threads or reduce the off-chip memory accesses. However, the usage of shared memory can limit the concurrent number of threads, therefore reducing the thread level parallelism (TLP). We observe that the usage of shared memory on current GPGPUs is too conservative, and one thread block should be able to release the shared memory as long as the shared memory is not used. We present three software approaches and one hardware approach to address this issue [87]. Our software approaches allow multiple thread blocks to share the same copy of shared memory by combining these thread blocks into one large thread block. Our hardware approach, on the other hand, introduces runtime allocation and de-allocation instructions for shared memory so that the shared memory can be released after the completion of the shared memory usage instead of the completion of a thread block. In the experiment, we select benchmarks, which use shared memory intensively, from AMD SDK [4][56], NVIDIA SDK [52], and GPGPUUSim [9]. Our software approaches achieves up to 1.95 times speedup and 1.44 times average speedup on GTX 285. Our hardware approach achieves up to 2.34 times speedup and 1.53 times average speedup based on simulation results.
Finally, the trend toward energy efficiency has led to the fused architectures, in which CPUs and GPUs are integrated onto the same chip. In the fused architecture, CPUs and GPUs share the off-chip memory and eliminate the memory transfer between CPUs and GPUs. We model such architecture by integrating a GPU simulator and a CPU simulator with shared L3 cache and off-chip memory. Then we propose a novel way to collaboratively utilize the CPU and GPU resources, called CPU-assisted GPGPU [85]. When a GPU kernel is launched on the GPU, we also start one pre-execution program on the CPU, which is generated using the GPU kernel. Since the CPU pre-execution executes only memory instructions and skips some workload, it can run ahead of the GPU program and fetch the data required by the GPU program in the L3 cache. As a result, the shared L3 cache is warmed up for the GPU program. Our performance results based on the simulation show up to 2.26 times speedup and on average 1.21 times speedup. Furthermore, we discuss other opportunities in the heterogeneous architectures.

1.2 Contributions

This dissertation makes several important contributions to many-core architectures. First, we present one compiler framework to improve the performances of naïve GPGPU programs by applying a set of optimization techniques. Our compiler takes one naïve GPGPU program as the inputs and generates optimized GPGPU programs. A set of scientific and media processing algorithms is used in the experiment. The experiment results show that our compiler can effectively improve the GPGPU programs without the effort of application developers.

Secondly, we develop one source to source compiler to automatically identify parallelizable code regions, insert LEO clauses, and optimize memory transfer between CPU and MIC, therefore improving the overall performances of MIC programs. The experiments with SpecOMP and NAS Parallel benchmarks demonstrate the effectiveness of our compiler.

Thirdly, since the usage of shared memory can limit the TLP and then hurt the performances of GPU programs, we present our solutions to enable shared memory multiplexing including three software approaches, which can work on existing GPGPUs, and a hardware approach, which can be integrated into future generation GPGPUs. The performance results show that our software approaches can achieve significant speedups on both NVIDIA GTX 285 and GTX 480, and our hardware solution implemented in a GPGPU simulation is effective with low hardware cost.
Finally, using prefetching as one example, we demonstrate that on fused heterogeneous architecture, on which the CPU and GPU are integrated into same chip with shared last level cache, the CPU prefetcher can significantly improve the throughput of GPU programs.

1.3 Outlines

The dissertation is organized as follows. In chapter 2, we will give a brief introduction of many-core architecture using the GPGPU and Intel MIC as examples, as well as the motivation of this dissertation. Chapter 3 presents our GPGPU compiler for memory optimization and parallelism management. We propose one source to source compiler to optimize Intel MIC programs in Chapter 4. Chapter 5 addresses the problem of too conservatively using shared memory on GPUs. Our proposed CPU-assisted GPGPU on fused CPU-GPU architecture is discussed in Chapter 6. Finally, Chapter 7 concludes the dissertation.
Chapter 2

Background

In this chapter, we first give a brief introduction about the architecture of GPGPU as one example of many-core architecture in Section 2.1. Section 2.2 discusses another many-core prototype Intel MIC. Section 2.3 presents the CPU-GPU fused architecture. We briefly discuss the related work in Section 2.4.

2.1 GPGPU Architecture

![Figure 2.1. Overview of GPGPU architecture](image-url)
The discrete GPGPUs work as coprocessors attached to CPUs through PCI Express bus as shown in Figure 2.1. Instead of operating on the system memory directly, each GPGPU has own off-chip memory system. If the GPGPU needs to access the data in the system memory, the data has to been first transferred from system memory to GPGPU memory, and vice versa. In order to accommodate both general purpose applications and graphics applications, the GPGPU off-chip memory is partitioned into different memory regions including global memory, constant memory, texture memory and local memory. Each memory region can be accessed by GPGPU cores through different cache hierarchies. Since the global memory is both readable and writeable, and supports large memory space, it is studied mostly in this dissertation. To access the off-chip memory efficiently, both memory latency and memory bandwidth have to been considered carefully. To achieve the high off-chip memory bandwidth, the vectorization as well as memory coalescing are two important techniques for GPUs. To hide the long latency off-chip memory, each GPU can support thousands of threads running concurrently. These threads running on a GPU follow the single-program multiple-data (SPMD) program execution model and are also organized in a hierarchical manner. Multiple threads are firstly grouped into a thread block, and then multiple thread blocks are grouped into a grid. Therefore a GPU kernel is launched to a GPU with a grid of thread blocks (TBs) using the NVIDIA CUDA terminology, which are called workgroups in OpenCL.

A GPU has a number of streaming multiprocessors (SMs) (15 SMs in a GTX 480). AMD/ATI HD 5870 GPU has similar architecture and has 20 SIMD engines (i.e., SMs), which is called Compute Unit in AMD latest Southern Island architecture. As shown in Figure 2.2, each SM contains multiple streaming processors (SPs) (32 in a GTX 480) for NVIDIA GPUs, and each SIMD has 16 stream cores (i.e., SPs) for AMD HD 5870. Threads in a TB form multiple 32-threads warps on NVIDIA GPUs or 64-threads wavefronts on AMD GPUs with each warp/wavefront being scheduled by the warp scheduler [24] and executed on these SPs or stream cores in the single-instruction multiple-data (SIMD) manner. Except caches for different memory regions, the on-chip memory resources include register files (128kB per SM in GTX 480, and 256kB per SIMD in HD 5870), and shared memory (16kB or 48kB on GTX 480 depending on configuration, and 32kB per SIMD in HD 5870). On NVIDIA GTX 480, the shared memory and L1 cache share the configurable 64K Bytes on-chip memory. Therefore, based on the application characterizations, it can be configured as 48K Bytes shared memory and 16K Bytes L1 cache, or 16K Bytes shared memory and 48K Bytes L1 cache. Preview studies [23][25][35][86][91] reveals the fact that the hardware-specific optimizations are
critical to the application development. To relieve the workload of application developers, we develop one source to source compiler which can automatically apply a set of optimization techniques based on the aforementioned hardware structure. As a result, the application developers only need to focus on identifying fine-grain thread-level parallelism without proficient understanding of GPU architecture. This work is presented in Chapter 3.

Current GPUs highly rely on TLP [80] to overcome the data dependency, control hazard and the off-chip memory latency. However the number of concurrent threads (i.e. TLP) can be limited by the aggressive usage of on-chip resources, such as the register file and shared memory. Among the on-chip resources, the shared memory works as a software-managed cache and plays an important role in optimizing performance of the applications. Application developers use shared memory mainly in three ways: 1) data communication among threads in a thread block as used in the Fast Fourier transform (FFT) [4]; 2) memory coalescing and data cache as used in Matrix Multiplication [56]; 3) array access as used in MarchingCubes [56]. If we ignore the performance impact, all three types of shared memory usage can be replaced with either global memory usage or local memory usage. For example, first the data communication can be implemented through the global memory with correct barrier instructions. Secondly, the un-coalesced memory access doesn’t affect the correctness of the
program. Therefore the shared memory usage for coalesced memory access can be removed by using un-coalesced memory access. Thirdly, as the alternative to shared memory, either local memory or global memory can be accessed as one array using array index. As off-chip memories, local memory and global memory, have long access latency. But they can be cached using the hardware-managed L1 cache to improve the performance. To investigate the benefit of the shared memory, we remove the shared memory usage from seven benchmarks and report the execution time of each benchmark in Figure 2.3. The detailed methodology and benchmark descriptions can be found in Section 5.3. Since the execution time in Figure 2.3 is normalized to the original execution time of each benchmark using shared memory, we can see that the shared memory is effective for almost all the benchmarks except MC. In Chapter 5, we will further characterize the share memory usage of each benchmark and discuss how we can improve the performance of the GPU program, which uses the shared memory intensively and therefore has the limited TLP.

![Figure 2.3. The performances of GPU programs after removing the share memory usage](image_url)
In order to accommodate the GPGPU architecture, the CUDA programming model is proposed to release the capability of GPGPUs to application developers. Next, we use one code example to illustrate the CUDA programming model as shown in Figure 2.4. First each GPU program consists of two parts: the host program running on a CPU and the device function, also called kernel function running on a GPU. Since a GPU works as a coprocessor, i.e. a GPU program cannot run standalone, we need to setup the GPU program using a CPU program. In the Figure 2.4a, the CPU program first allocates three memory arrays on the GPU using the first three lines. Then in the line 4 and 5, the data in the array A and B that are allocated in CPU memory, will be transferred to d_A and d_B in GPU memory, respectively. In line 6 of Figure 2.4a, the GPU kernel function is called with thread block configurations and function. Finally, the result of GPU computing is transferred back to CPU memory in line 7. We also show the kernel function in Figure 2.4b, in which each thread first computes it thread id in line 3 and performs the addition operation in line 4.

![CUDA code example](image)

**Figure 2.4. CUDA code example**
2.2 Intel MIC

Many Integrated Core Architecture (MIC), officially named as Xeon Phi, is released as the successor of Intel Larrabee and targeted at the supercomputing market. As a competitor of GPGPUs, Intel MIC is also capable of teraflops computation power and hundreds of gigabytes memory bandwidth by employing the many-core architecture. Similar to GPGPUs, Intel MIC has own device memory, and it is also connected to the CPU through the PCIE bus, therefore requiring explicit memory transfer between a CPU and a MIC. The microarchitecture of Intel MIC, however, is more close to the traditional CPUs rather than GPUs, even though each MIC device has many in-order cores as GPGPUs. The core of a MIC is very different from the SM of a GPU. First, each MIC core has been deployed a Vector Processing Unit (VPU) and 32 512-bit vector registers. New 512-bit wide SIMD vector instructions are supported to utilize the VPU, while GPUs explore the warp/wavefront level execution to deliver multiple flops in each cycle. Secondly, each SM of a GPU can support thousands of threads, and high level TLP is preferred to achieve high performance. But a MIC core can support only four hardware threads, and therefore single thread optimization is more critical. Thirdly, Intel MIC relies on the large shared L2 cache (up to 32M bytes) for the efficient utilization of off-chip memory bandwidth, while GPUs have smaller L2 cache (768K bytes on NVIDIA GTX 480) and employ techniques including data vectorization and memory coalescing, to achieve high memory bandwidth.

```
#pragma offload target(mic) in(A,B:length(n)) out(C:length(n))
{
#pragma omp parallel for shared(A,B,C) private(i)
  for (i=0; i<n; i++){
    C[i] = B[i] + A[i];
  }
}
```

**Figure 2.5 Code example for Intel MIC**

Next we briefly introduce the programming model of Intel MIC also using Vector-Add as shown in Figure 2.5. Compared with the openmp program, the difference of the code in Figure 2.5 is the
pragma started with “#pragma offload”. Such pragma indicates that the code section will be executed in the MIC. The in clause in the pragma means the array A and B need to be copied to MIC device, and the size of data to be transferred is specified by length keyword. In contrast, the out clause is used to copy data from MIC device to CPU memory. The parallelization of the MIC program relies on the openmp programming model so as to minimize the change of legacy code. In order to adapt legacy programs to the MIC and utilize the MIC efficiently, we need to answer following questions: 1) which code section should be selected to be executed on MIC device; 2) how can we generate the offload pragma automatically; 3) how we optimize the memory transfer between the CPU and the MIC. As will be seen in Chapter 4, we address these issues by developing our compiler framework.

2.3 CPU-GPU Fused Architecture

In order to achieve the energy efficiency, instead of building high performance GPGPUs, another trend is to integrate a GPU and a CPU into single chip. In such fused architectures including AMD accelerated processing unit (APU) and Intel Sandy Bridge, a CPU and a GPU share the same off-chip memory, and therefore the memory transfer between the CPU and the GPU can be eliminated. In AMD APU architecture, the GPU has the same microarchitecture of discrete GPGPUs, but has reduced number of Computer Units (CUs) running at the lower frequency to save area and energy consumption. Intel Sandy Bridge provides a unified last level L3 cache, which is not available in AMD APU. But current GPU of Intel Sandy Bridge doesn’t support either CUDA or OpenCL languages for general purpose computation. In Chapter 6, we propose CPU-assisted GPGPU based on our modeled fused CPU-GPU architecture, in which the CPU and GPU share both last level cache and off-chip memory.

2.4 Related Work

The many-core architectures have attracted researchers in different areas. Previous work can be classified in three categories: application optimizations, compiler developments, and architectural improvements. First, the application optimizations [2][25][35][37][42][47][65][66][67][77][90] focus on the performance improvement for specific applications on many-core platforms. However, these optimizations have to be applied case by case, and it requires lots of effort of application developers. By using compiler techniques, some optimization techniques [7][11][12][15][44][45][71][76][89][91] can be automatically used to improve the applications without the effort of application developers. While the application optimization and compiler optimization can work on existing hardware, the
overhead of these software approaches is not negligible. Therefore, many hardware solutions [9][24][41][79][81] are also presented to improve the many-core architectures. We also can classify previous work based on the problem domains. Some major research topics include the memory optimization [10][35][37][41][76][79], divergence elimination [24][89] and CPU-GPU heterogeneous architecture [13][21][34][47][79]. In this dissertation, we present our compiler solutions in Chapter 3 and Chapter 4, and our architecture improvements in Chapter 5 and Chapter 6. We leave the detailed discussion for related work in each chapter.
Chapter 3

A GPGPU Compiler for Memory Optimization and Parallelism Management

3.1 Introduction

The high computational power and affordability of state-of-art graphics processing units (GPU) have made them the first widely accessible parallel computers with teraflops capability. To fully realize the power of general purpose computation on graphics processing units (GPGPU), two key issues need to be considered carefully: (1) how to parallelize an application into concurrent work items and distribute the workloads in a hierarchy of thread blocks and threads; and (2) how to efficiently utilize the GPU memory hierarchy, given its dominant impact on performance. As these two issues usually coupled together and finding an optimal tradeoff between different levels of parallelism and memory optimizations requires detailed understanding of GPU hardware, developing high performance GPGPU programs remains challenging for application developers. Furthermore, GPU hardware architectures are evolving rapidly, which makes the code developed and tuned for one generation (e.g., NVIDIA GTX 8800) less optimal for the next one (e.g., NVIDIA GTX280). Our envisioned solution to these problems is to let application developers identify fine-grain thread-level parallelism and/or data-level parallelism and to use an optimizing compiler to perform memory and parallelism optimizations. This way, we leverage the algorithm-level expertise of application developers and at the same time relieve them of low-level hardware-specific performance optimizations.

Our compiler works as follows. The input is a naïve GPU kernel function, which is functionally correct but does not include any device-specific performance optimizations. Such a kernel function represents the user-identified fine-grain work item that can run concurrently. A typical example of a
fine-grain work item is the computation of a single data element in the output domain. The compiler analyzes the naïve kernel, checks the off-chip memory access patterns, and optimizes the memory accesses through vectorization and coalescing to achieve high data access bandwidth. Then the compiler analyzes data dependencies and identifies possible data sharing across threads and thread blocks. Based on data sharing patterns, the compiler intelligently merges threads and/or thread-blocks to improve memory reuse through the register file and the on-chip shared memory. These merges provide a novel way to achieve loop tiling and unrolling by aggregating fine-grain work items into threads and thread blocks. Additionally, the compiler schedules the code to enable data prefetching so as to overlap computation with memory access latencies. To avoid partition camping [67] (i.e., to distribute memory traffic evenly across memory partitions), thread blocks are checked for their memory accesses and depending on the thread block dimensions, either an address offset is inserted or the block identifiers (ids) are remapped, if necessary. The compiler also performs hardware-specific tuning based on hardware parameters such as the register file size, the shared memory size, and the number of cores in the target GPU.

Besides the aggressive compiler optimizations, another distinguishing feature of our compiler is that the optimized code is reason-ably understandable compared to the code generated using algebraic frameworks such as polyhedral models [61]. As a result, it is relatively easy to reason about the optimized code generated by our compiler, which facilitates algorithm-level exploration.

In our experiments, we used the compiler to optimize 10 scientific and image processing functions. The experimental results on NVIDIA 8800 GTX and NVIDIA GTX 280 GPUs show that our optimized code can achieve very high performance, either superior or very close to the NVIDIA CUBLAS 2.2 library and up to 128X over the naïve implementation.

In summary, our work makes the following contributions. (1) We propose a compiler for GPGPU programming that enables the application developers to focus on algorithm-level issues rather than low-level hardware-specific performance optimizations. (2) We propose a set of new compiler optimization techniques to improve memory access bandwidth, to effectively leverage on-chip memory resource (register file and shared memory) for data sharing, and to eliminate partition conflicts. (3) We show that the proposed optimizing compiler is highly effective and the programs optimized by our compiler achieve very high performance, often superior to manually optimized codes.
The remainder of the chapter is organized as follows. In Section 3.2, we present a brief background on the NVIDIA CUDA programming model [51] and highlight key requirements for high performance GPU computation. In Section 3.3, we present our proposed optimizing compiler in detail. Section 3.4 explores the design space of our proposed optimizations. A case study of matrix multiplication is presented in the Section 3.5 to illustrate the compilation process. The experimental methodology and results are presented in the Section 3.6. In Section 3.7, we highlight the limitations of the proposed compiler. Related work is discussed in Section 3.8. Finally, Section 3.9 concludes our chapter and discusses future work.

### 3.2 Background

In this section, based on the GPGPU background introduced in Chapter 2, we summarize the key aspects for high performance GPGPU code as they are the main focus of our proposed compiler optimizations.

1. **Off-chip memory access bandwidth.** To utilize the off-chip memory bandwidth efficiently, memory accesses need to be coalesced and each data item may need to be a vector type, depending on specific GPU hardware. Memory coalescing refers to the requirement that the accesses from 16 consecutive threads in a warp (i.e., a half warp) can be coalesced into a single contiguous, aligned memory access [51]. In this chapter, we refer to such a coalesced contiguous, aligned region as a coalesced segment. If each memory access is of the type ‘float’, each segment starts from an address which is a multiple of 64 bytes, and has the size of 64 bytes. The memory bandwidth utilization may be significantly improved when each of coalesced memory accesses is of a vector data type, such as float2 (a vector of two float numbers) and float4 (a vector of four float numbers). For ATI/AMD HD 5870, the sustained bandwidth reaches 71GB/s, 98GB/s, and 101GB/s when accessing 128MB data using the float, float2, and float4 data types, respectively. In comparison, for the same data transmission on NVIDIA GTX 280, the sustained bandwidth is 98GB/s, 101GB/s, and 79GB/s using the float, float2, and float4 data types, respectively.

2. **Shared memory.** The common usage of shared memory is a software-managed cache for memory reuse. Although it has low access latencies, shared memory is slower than register files and has certain overheads beyond access latency. First it needs to be synchronized to ensure proper access order among the threads in a thread block. Secondly, the shared memory in NVIDIA GPUs has 16 banks, and bank conflicts can impair the performance.
3. Balanced resource usage. As multiple threads in the same thread block and multiple thread blocks compete for limited resources in an SM, including the register file, the shared memory, and the number of the thread contexts being supported in hardware, we need to carefully balance parallelism and memory optimizations.

4. Off-chip memory partitions. In current GPUs, off-chip memory is divided into multiple partitions. There are 6 and 8 partitions in GTX8800 and GTX280, respectively, and the partition width is 256 bytes. To use the partitions effectively, the memory traffic should be evenly distributed among all the partitions. Otherwise, the requests may be queued up at some partitions while others are idle. This is referred to as partition camping [67] or partition conflicts, which are similar to bank conflicts at shared memory but incur much higher performance penalties. Since concurrent memory requests are issued on a per half-warp basis from all active thread blocks, partition conflicts happen across different thread blocks.

Note that the key performance issues listed above are not unique to current GPUs. Future many-core architectures will probably use similar approaches to achieve high memory bandwidth (i.e., coalescing, multiple memory partitions) and to reduce memory access latency (i.e., on-chip software managed cache or shared memory). So, the proposed compiler optimizations are expected to be relevant beyond the scope of GPGPU.

### 3.3 An Optimizing GPGPU Compiler

Our proposed compiler framework is shown in Figure 3.1. The input to our compiler is a naïve GPU kernel function, which is functionally correct, but does not include any device-specific performance optimizations. For many scientific computing and media processing functions, the naïve version is simply the code to compute one element/pixel in the output matrix/image. Typically such code is straightforward to extract from the sequential CPU code. One common example is the loop body from a heavily executed loop. In Figures 3.2a and 3.2b, we show the sample naïve kernel functions for the matrix multiplication (mm) and matrix-vector multiplication (mv) algorithms, respectively. Each computes one element at the position (idx, idy).

In Figure 3.2, ‘idx’ and ‘idy’ are the position/coordinate of the element in the output matrix. In the CUDA programming model, ‘idy’ can be viewed as the absolute thread id along the Y direction, which is equal to (blockIdx.y*blockDimy + threadIdx.y) in the CUDA code. Correspondingly, ‘idx’ is the absolute thread id along the X direction, which equal to (blockIdx.x*blockDimx + threadIdx.x).
In comparison, the CUDA predefined ‘threadIdx.x’ and ‘threadIdx.y’ are the relative thread position/coordinate within a thread block and we refer to them as ‘tidx’ and ‘tidy’ for short. Both tidx and tidy are independent of the thread block ids.

As can be seen from the two examples, the naïve kernel functions don’t have any shared memory usage and do not require thread block partition. In other words, we may simply assume every block only has one thread. All the arrays are initially in the off-chip global memory.

For applications which require synchronization among computing different output pixels, e.g., reduction operations, a global sync function is supported in the naïve kernel.

**Figure 3.1. The framework of the proposed compiler**

To facilitate compiler optimizations, the following (optional) information can be conveyed using the ‘#pragma’ interface: the size of the input and output dimensions, and the output variable names. The
latter can be used to eliminate global memory writes to temporary variables when they are moved to shared memory.

Given the naïve kernel function, the compiler takes the following steps to generate the optimized kernel code. First, depending on the targeted GPUs, the compiler attempts to group memory accesses into vector data accesses. Secondly, the off-chip memory accesses are checked to see whether they satisfy the requirements for memory coalescing. If not, the code will be converted to coalesced memory accesses using shared memory as temporary storage. Thirdly, the compiler analyzes data dependencies and sharing patterns to determine how the data are shared among the neighboring thread blocks. Based on data sharing patterns, the compiler merges both threads (i.e., combining several threads in different thread blocks into one) to enable the data reuse through registers and thread blocks (i.e., combining several blocks into one) to increase data reuse through shared memory. The data reuse information is also used to disable certain memory coalescing transformations when there is little or no data reuse. After thread/thread-block merge, the compiler schedules the code to perform data prefetching. Then, the compiler checks the memory accesses from different thread blocks for partition camping and either inserts address offsets or remaps thread block ids, if necessary. Finally, the compiler generates the optimized kernel and the parameters (i.e., the thread grid & block dimensions) to invoke the kernel function.

The optimization process described above can also be used as a generic methodology to guide manual optimizations of GPGPU programs. As a result, our optimized code is reasonably understandable, as will be seen in the remainder of Section 3.3.

### 3.3.1 Vectorization of Memory Accesses

As discussed in Section 3.2, the data type of memory accesses may have significant impact on bandwidth utilization. Therefore, the compiler first checks data accesses inside the kernel function to see whether they can be grouped in a vector type data access. Since different GPUs feature significantly different requirements on vector types for bandwidth utilization, the compiler follows different rules to adjust the aggressiveness of vectorization. In this chapter, we focus on CUDA and NVIDIA GPUs, in which a vector of two floats (i.e. float2) is the preferred data type but the bandwidth improvement over the float type is less than 3%. Therefore, we use the following strict rule: if there is a pair of accesses to the same array with the indices: 2*idx+N and 2*idx+N+1, where N is an even number, the compiler generates a float2 variable \( f2 \) with array offset as idx+N/2 and
replaces the original array accesses with \(f2.x\) and \(f2.y\). This rule is essentially designed for applications using complex numbers when the real part is stored next to the imaginary part of each data element. Note that this vectorization of data accesses is simpler than classical vectorization.

For AMD/ATI GPUs, due to the much more profound impact on bandwidth, the compiler is more aggressive and also groups data accesses from neighboring threads along the X direction into float2/float4 data types. The tradeoff of vectorization of data accesses is that if the vector data accesses are not coalesced (Section 3.3.2) and the compiler converts them into coalesced ones (Section 3.3.3) through shared memory, there may be bank conflicts. For AMD/ATI GPUs, the benefits of vectorization far outweigh the penalties of shared memory bank conflicts. For NVIDIA GPUs, however, the benefits from vectorization are limited. Therefore, the compiler skips these additional steps to vectorize data accesses.

![Figure 3.2. Examples of naive kernel functions.](image)

**Figure 3.2. Examples of naive kernel functions.**

### 3.3.2 Checking Memory Coalescing

As discussed in Chapter 2, GPGPU employs the SPMD model and the threads in a single warp execute the kernel function in the SIMD mode. Therefore, in order to determine whether off-chip memory accesses can be coalesced, we need to compute the addresses of each memory access in the kernel function for different threads. As arrays are the most common data structure in scientific and media processing, we consider four types of array indices and affine transformations of these indices:

1. **Constant index**: the constant value is used in an array index, for example, the constant integer ‘5’ in ‘\(a[idy][i+5]\)’. 
2. **Predefined index**: the predefined numbers, such as absolute thread ids, idx, idy, and relative thread ids, tidx (i.e., threadIdx.x), tidy (i.e., threadIdx.y), are used as an array index. For example, ‘idy’ in ‘a[idy][i+5]’.

3. **Loop index**: a loop iterator variable is used as an array index, for example, ‘i’ in ‘b[i][idx]’ in Figure 3.2a.

4. **Unresolved index**: an array index is used, which is not one of the first three types. For example, an indirect access ‘a[x]’ where ‘x’ is a value loaded from memory. As our compiler cannot determine the addresses of such indices, we simply skip them without checking whether they can be coalesced.

Among the four types of indices, the addresses corresponding to the first two are fixed for a given thread. For the third, however, we need to check different values of the loop iterator. Assuming that a loop index starts from S with increment Incr, then we need to check the index addresses from the first 16 iterations: S, S+Incr, S+2*Incr, to S+15*Incr. The reason is that the same behavior repeats for remaining iterations in terms of whether the access can be coalesced as the difference in addresses is a multiple of 16.

After determining the types of array indices in the kernel function, for each memory access instruction, the compiler computes the addresses from the 16 consecutive threads in the same warp (i.e., a half warp) to see whether they can be coalesced. As discussed in Section 3.2, if we assume the array type of ‘float’, the coalesced accesses will form a coalesced segment, which starts from an address, whose value is a multiple of 64, and has the size of 64 bytes. Among the addresses from the 16 threads, we refer to the smallest one as the ‘base address’. The differences between the base address and the addresses from the subsequent 15 threads are referred to as ‘offsets’. To satisfy the coalescing requirement, the base address needs to be a multiple of 64 and offsets need to be 1 to 15 words. The following two rules are used to handle common array accesses.

For an index to a multi-dimensional array, e.g., ‘A[z][y][x]’, the index to the higher-order dimensions, e.g., the ‘y’ and ‘z’ dimensions, should remain the same for all the 16 threads in the half warp. Otherwise, for example, if the predefined index ‘idx’ (the thread id along the ‘x’ direction) is used in an index to the ‘y’ dimension in a multi-dimension array ‘A[][idx][0]’, the accesses from the 16 threads will be ‘A[][0][0]’, ‘A[][1][0]’, ‘A[][2][0]’, etc., and are not coalesced.
When a loop index is used in the kernel function, the compiler computes the base address and the offsets for each possible value of the loop iterator. For example, for the address ‘a[idy][i]’ in Figure 3.2a, the base address is ‘&a[idy][0]’ when the iterator ‘i’ is 0; ‘&a[idy][1]’ when ‘i’ is 1, etc. The offsets are all zeros as the addresses do not change for different threads in the same half warp. As both the base addresses and the offsets do not meet the condition, the array access ‘a[idy][i]’ is not coalesced. For the array access ‘b[i][idx]’ in Figure 3.2a, the base address is ‘&b[0][0]’ when ‘i’ is 0; ‘&b[1][0]’ when ‘i’ is 1, etc.. The offsets are from 1 word to 15 words. Thus, the array access ‘b[i][idx]’ is coalesced as long as each row of array b is aligned to the multiple of 16 words. For the array access ‘b[i][idx+i]’, although the offsets satisfy the condition for every possible ‘i’, it is not a coalesced access since the base address is not always a multiple of 16 words, e.g., ‘b[1]’ when ‘i’ is 1.

### 3.3.3 Converting Non-Coalesced Accesses into Coalesced Ones

After the compiler analyzes every array access in the kernel code, the compiler converts the non-coalesced accesses into coalesced ones through shared memory. The observation here is that for each non-coalesced memory access instruction, the compiler can determine the coalesced segments that contain the data required by the non-coalesced memory accesses from the half warp. The compiler then introduces shared-memory array variables, inserts statements (coalesced memory accesses) to initialize the shared memory variables, and replaces the original global memory accesses with shared memory accesses. The thread block size is also set to 16 so that each thread block contains one half warp. The ‘syncthreads’ function is also inserted to ensure the proper access order.

For array accesses using constant or predefined indices, the process is typically straightforward. For example, the non-coalesced access, ‘A[idy][0]’, the coalesced segment is ‘A[idy][0:15]’. The compiler inserts a shared-memory array variable ‘sA[0:15]’ and initializes the ‘sA[0:15]’ with ‘A[idy][tidx]’, where tidx is relative thread id within the warp. In the case when ‘idx’ is used in an index to a multi-dimensional array, the compiler may introduce a loop to load the required data for a half warp. For example, for an array access ‘A[idx][0]’, the required data for a half warp is ‘A[(idx-tidx)+(0:15)][0]’, where ‘(idx-tidx)’ provides the start address of each thread block, which is the same as the start address of the half warp as each thread block only contains a half warp at this time. The coalesced segments that contains the required data are ‘A[(idx-tidx)+(0:15)][0:15]’. In the introduced loop of 16 iterations, a shared memory array is initialized with ‘A[(idx-tidx)+l][tidx]’, where l is the iterator of the newly introduced loop. From these examples, it can be seen that not all the data loaded in the shared memory are useful, the compiler will perform data reuse analysis (Section 3.3.4) to
determine whether this transformation is beneficial or not. If it is not, the compiler will skip coalescing transformation on this access. In the special case where an array access involves both ‘idx’ and ‘idy’, such as ‘A[idx][idy]’, the compiler analyzes the feasibility to exchange ‘idx’ and ‘idy’ to make it coalesced. This transformation is equivalent to loop interchange on the CPU code.

For array accesses using a loop index, ‘A[m*i+n]’, where ‘i’ is the loop iterator and m and n are constants, the compiler unrolls the loop for 16/(GCD(m,16)) times if m is less than or equal to 8. If m is greater than 8, the coalesced access has little benefit due to limited reuse across different iterations. Then, the compiler groups the accesses from unrolled loops into coalesced ones. For example, for the array access ‘A[idy][i]’ where ‘i’ is the loop iterator, the segment ‘A[idy][0:15]’ contains all the required data for the first 16 iterations. The compiler unrolls the loop for 16 times, introduces shared memory variable sA[0:15] which are initialized with A[idy][tidx+i] (coalesced as the increment of ‘i’ is 16 after unrolling), and replaces ‘A[idy][i]’ with ‘sA[i]’.

**Figure 3.3. Coalesced kernels generated by the compiler.**
For the naïve kernels in Figure 3.2, the coalesced versions are shown in Figure 3.3. The inner loop with the iterator ‘k’ is a result of unrolling the outer loop with the iterator ‘i’. In the naïve kernel in Figure 3.2a, the access ‘a[idy][i]’ is not coalesced, which results in loop unrolling as described above. ‘b[i][idx]’ is coalesced and it transforms to ‘b[(i+k)][idx]’ due to unrolling for ‘a[idy][i]’. In the mv kernel in Figure 3.2b, both accesses ‘a[idx][i]’ and ‘b[i]’ are not coalesced. Converting the access ‘b[i]’ into coalesced accesses involves a loop unrolling of 16 (=16/GCD(1,16)) times and it becomes ‘b[i+tidx]’ in Figure 3.3b. For the access ‘a[idx][i]’ the loop with the iterator ‘l’ is introduced and the access is transformed to ‘a[(idx-tidx)+i][i+tidx]’. In addition, the compiler may add padding to the shared memory arrays to avoid bank conflicts and padding to input data arrays to ensure that the row size of each array is a multiple of 16 words so as to meet the requirement of memory coalescing.

After memory coalescing, the kernel code generated by our compiler has the following characteristics:

1. Each thread block has 16 consecutive threads (i.e., only a half warp) along the X direction, because 16 threads are needed by hardware to coalesce memory accesses and they communicate with each other through shared memory. The number of threads in each thread block will be expanded during the next optimization phase (Section 3.3.5) to make sure there are enough threads in each thread block.

2. There are two types of global memory load statements: (a) Global memory to shared memory (G2S): the statements read data from global memory and store them into the shared memory, such as (S2) in Figure 3.3a. (b) Global memory to register (G2R): the statements read data from global memory and save them to registers. For example, in (S5) in Figure 3.3a, the global memory access ‘b[(i+k)][idx]’ loads the data into registers.

### 3.3.4 Data Dependencies and Data Sharing

In this step, the compiler detects data dependency and data sharing. Such analysis is similar to those used in analyzing affine array accesses for locality optimization and parallelization [1]. As our compiler has already enforced memory coalescing by associating coalesced segments with each global memory access, the compiler can detect data sharing by comparing whether the address ranges of the segments have overlaps. In the applications that we studied, we found that data sharing happens most frequently among neighboring blocks along the X or Y direction. Therefore, our current
compiler implementation mainly focuses on checking data sharing among neighboring thread blocks and also the thread blocks with a fixed stride along the X or Y direction.

![Diagram of thread block merging](image)

**Figure 3.4.** Improve memory reuse by merging neighboring thread blocks.

```c
int i = 0;
float sum = 0;
for (i=0; i<w; i=(i+16)) {
    __shared__ float shared0[16];
    if (tidx<16) { /*inserted due to block merge to remove redundant loads */
        shared0[(0+tidx)]=a[idy][(i+tidx)+0]);
    }
    __syncthreads();
    int k;
    for (k=0; k<16; k=(k+1)) {
        sum+=shared0[(0+k)]*b[(i+k)][idx]);
    }
    __syncthreads();
}c[idy][idx] = sum;
```

**Figure 3.5.** The kernel function for matrix multiplication, after merging blocks along the X direction.
The data sharing/reuse information is also used to determine whether the code conversion for memory coalescing is beneficial. As described in Section 3.3.3, shared memory is used as temporary storage to achieve memory coalescing. The data in the shared memory, however, may not be useful as they are simply loaded from off-chip memory to satisfy the coalescing requirement. For example, the compiler loads $A[idy][0:15]$ in order to convert the access $A[idy][0]$ into a coalesced one. Currently, our compiler employs a simple rule to check whether an access needs to be converted: if the loaded data in shared memory have no reuse, it is not converted. A more crafted heuristic may further rank code conversions for different accesses by comparing their shared memory usage and number of data reuses, and then select the most beneficial ones if shared memory is used up. We left such investigation as our future work to refine our compiler framework.

### 3.3.5 Thread/Thread-Block Merge to Enhance Memory Reuse

![Figure 3.6. Improve memory reuse by merging threads from neighboring thread blocks.](image)

After detecting that there exists data sharing among thread blocks (mainly neighboring blocks), we propose two new techniques to enhance data sharing so as to reduce the number of global memory
accesses: merging thread blocks and merging threads. Thread-block merge determines the workload for each thread block while thread merge decides the workload for each thread. These two techniques combined are essentially a way to achieve loop tiling and unrolling by aggregating the fine-grain work items into threads and thread blocks. We first present the two techniques and then discuss how compiler prioritizes one over the other.

### 3.3.5.1 Thread-block merge

When our compiler determines that multiple thread blocks share some common data, it may choose to merge them into one thread block, as shown in Figure 3.4.

To illustrate the procedure to merge thread blocks, we show how our compiler combines two neighboring blocks along the X direction into one. First, the compiler re-computes the thread id information within the thread block (i.e., tid). As two thread blocks along the X direction are merged, idx, idy and tidy remain the same while tidx is re-computed as \( \text{idx} \% (N \times \text{blockDim.x}) \), where \( N = 2 \) for Figure 3.4. Secondly, for the statements that result in data sharing, we add control flow to ensure that the global memory data are loaded only once. For the matrix multiplication example in Figure 3.3a, the statement S2 in threads from two neighboring thread blocks accesses the same segment. Therefore, we add an ‘if (tidx < blockDim.x)’ statement to eliminate redundant global memory accesses, as shown in Figure 3.5. Thirdly, the thread block dimension is resized (blockDim.x = 2*blockDim.x).

As thread-block merge determines the workload for each thread block and all threads in the same thread block reuse data in shared memory, it essentially achieves loop tiling for locality and parallelism optimizations.

### 3.3.5.2 Thread merge

The other approach to enhance data sharing is to merge threads from different thread blocks, which combines several threads’ workloads into one, as shown in Figure 3.6. Compared to thread-block merge, after these threads are combined into one, they can share not only shared memory, but also the registers in the register file. Furthermore, some control flow statements and address computation can be reused, thereby further reducing the overall instruction count. The limitation is that an increased workload typically requires a higher number of registers, which may reduce the number of active threads that can fit in the hardware. From the discussion, it can be seen that thread merge achieves the
effects of loop unrolling. Note that thread merge also combines multiple thread blocks into one but it
does not increase the number of threads in each thread block.

```
int i = 0;
float sum_0 = 0;
......
float sum_31 = 0;
for (i=0; i<w; i=(i+16)) {
    __shared__ float shared0_0[16];
    ......
    __shared__ float shared0_31[16];
    if (tidx<16) {
        /* 32 is the number of the threads to be merged */
        shared0_0[(0+tidx)] = a[idy*32+0][((i+tidx)+0)];
        ......
        shared0_31[(0+tidx)] = a[idy*32+31][((i+tidx)+0)];
    }
    syncthreads();
    int k;
    for (k=0; k<16; k=(k+1)) {
        float r0 = b[(i+k)][idx];
        sum_0+=shared0[(0+k)]*r0;
        ......
        sum_31+=shared0_31[0+k]*r0;
    }
    __syncthreads();
}
c[idy*32+0][idx] = sum_0;
......
c[idy*32+31][idx] = sum_31;
```

**Figure 3.7. The matrix multiplication kernel after merging 32 threads in 32 adjacent blocks along the Y direction.**

To illustrate the procedure to merge threads, we show how our compiler combines \( N \) neighboring blocks along the Y direction into one. First, the compiler re-computes the thread id information. As we merge threads from two thread blocks along the Y direction, the absolute thread ID along the X direction ‘idx’ remains the same while the thread ID along the Y direction ‘idy’ will be changed to
idy*N, idy*N+1, idy*N+2..., idy*N+(N-1) for the N replicated statements. The thread id information within a thread block remains the same. Secondly, for the statement that results in data sharing, we need only one copy. Thirdly, for the control flow statement such as loops, we also only need one copy. Fourth, for the remaining statements including data declaration, ALU computation statement and other memory access statements, we replicate them for N times. For the matrix multiplication example in Figure 3.5, the array access ‘b[(i+k)][idx]’ results in the shared data among the thread blocks along the Y direction (as the access address is not dependent on ‘idy’). The compiler merges 32 neighboring blocks along the Y direction using thread merge, as shown in Figure 3.7.

### 3.3.5.3 Selection between thread merge and thread-block merge

As discussed in Section 3.3.3, the code generated by the compiler after memory coalescing has two types of global memory accesses: global to shared memory (G2S) and global to register (G2R). If data sharing among neighboring blocks is due to a G2S access, the compiler prefers thread-block merge to better utilize the shared memory. When data sharing is from a G2R access, the compiler prefers to merge threads from neighboring blocks due to the reuse of registers. If there are many G2R accesses, which lead to data sharing among different thread blocks, the register file is not large enough to hold all of the reused data. In this case, thread block merge is used and shared memory variables are introduced to hold the shared data. In addition, if a block does not have enough threads, thread-block merge instead of thread merge is also used to increase the number of threads in a block even if there is no data sharing.

### 3.3.6 Data Prefetching

Data prefetching is a well-known technique to overlap memory access latency with computation. To do so, the compiler analyzes the memory accesses in a loop and uses a temporary variable to prefetch data for the next iteration before the computation in the current loop. The process is illustrated in Figure 3.8. The code before insertion of prefetching is in Figure 3.8a and Figure 3.8b shows the code after insertion. Besides the temporary variable, additional checking is added to ensure that the prefetching access does not generate unnecessary memory accesses.

The overhead of data prefetching code is the increased register usage due to the temporary variables. If the register can be used for data reuse (e.g., as a result of thread merge), the compiler skips this optimization.
Figure 3.8. A code example to illustrate data prefetching.

3.3.7 Eliminating Partition Camping

In this step, the compiler reuses the address access patterns obtained for thread/thread-block merge to see whether they lead to partition camping. As neighboring thread blocks along the X direction are likely to be active at the same time, the compiler focuses on the addresses that involve blockIdx.x or bidx in short. Those accesses without involving bidx either access the same line in the same partition (e.g., A[0]) or access the same partition at different times (e.g., A[bidy][0] based on the assumption that thread blocks with different bidy will execute at different times). The following rules are followed by our compiler.

Partition Camping Detection: If an array access involves bidx, the compiler checks the address stride between the two accesses from the two neighboring blocks (i.e., one with block id bidx and the other with bidx+1). The compiler detects partition camping if the stride is a multiple of \((\text{partition size} \times \text{number of partitions})\). For example, for an array access \(A[\text{idx}]\), it is equivalent to \(A[\text{bidx} \times \text{blockDimx} + \text{tidx}]\). The stride between two neighboring blocks is blockDimx, whose value then decides whether there are partition conflicts (i.e., two concurrent accesses to the same partition).
**Partition Camping Elimination:** If an access results in partition conflicts, depending on how thread blocks are organized, we use two ways to eliminate partition conflicts:

1. If thread blocks are arranged in one dimension, we add a fixed offset, *(the partition width * bidx)*, to the access and update the loop bounds to accommodate the change. For example, in mv, the output is a vector. So the thread blocks are organized in one dimension. The accesses A[idx][i] (or the coalesced version A[((idx-tidx)+1)][(i+tidx)]), from neighboring thread blocks result in partition camping if the width of A is a multiple of *(partition size * number of partitions)*, as shown in Figure 3.9a. With the added offset, the access pattern is changed to Figure 3.9b, eliminating partition camping.

2. If thread blocks are organized in two or more dimensions, we apply the diagonal block reordering proposed in [67], which essentially changes the workload (or tile) that each thread block is assigned to. The diagonal mapping rule is newbidy = bidx and newbidx = (bidx+bidy) mod gridDim.x.

---

**Figure 3.9.** Eliminating partition camping. (a) Accesses to array A resulting in conflicts at partition 0. (b) Adding an offset as *(partition size * bidx)* eliminates the conflicts. The dark regions represent the memory footprint of A[idx][0] from different thread blocks.
3.4 Design Space Exploration

3.4.1 The Number of Threads in A Thread Block

In our compiler algorithm, the number of threads in a thread block is determined by thread/thread-block merge. The CUDA programming guide [55] suggests that one SM should have at least 192 active threads to hide the latency of register read-after-write dependencies. Because our compiler tries to use a number of resources (the shared memory due to thread-block merge and the register file due to thread merge) for better memory reuse, it is possible that the code after thread/thread-block merge requires a large amount of shared memory and registers so that one SM can only support a limited number of thread blocks. To balance the thread-level parallelism and memory reuse, our compiler tries to put 128, 256, or 512 threads into one thread block (equivalent to merging of 8, 16, and 32 blocks), if possible. Also, the compiler varies the degrees of thread merge (i.e., how many threads to be merged into one) across 4, 8, 16, or 32 so as to balance register-based data reuse and thread-level parallelism. As such, the combination of these design parameters creates a design space to explore. As discussed in Section 3.3.5, merging threads/thread blocks is one way to achieve loop tiling and unrolling. So, exploring such a design space is similar to finding the best tile size and unrolling factors for parallelization and locality enhancement. Due to the non-linear performance effect of those parameters on GPU performance, the compiler generates multiple versions of code and resorts to an empirical search by test running each version to select the one with the best performance. Another way is to use an analytical performance model [7] [30] to predict the performance of each version, but this requires higher accuracy than current models. Moreover, based on our experiments, the optimal version may be dependent upon the size of the input arrays, which implies that unless the compiler knows detailed information of the intended inputs, it is almost inevitable that the compiler must run multiple versions of code in order to find the optimal one.
3.4.2 Hardware Specification

GPU hardware is evolving rapidly. Although different generations of GPU hardware may share similar architecture, e.g., NVIDIA GTX8800 and GTX280, there are significant changes, e.g., the register file size, which may have a strong impact on performance. When there are more registers, more threads can be put into one block or one thread can use more registers for temporary data. As a result, an optimized code tuned for one GPU generation may not be optimal for the next. To solve this problem, our compiler generates different versions of optimized code based on different machine descriptions so that they can be deployed on different GPU platforms.

3.5 Case Study: Matrix Multiplication

Matrix multiplication (mm) is a commonly used algorithm and there has been continuing effort to improve its performance [77]. The fine-tuned implementation in NVIDIA CUBLAS 1.0 has a throughput of 110 GFLOPS when computing the product of two 2kx2k matrices on GTX 8800. In CUBLAS 2.2, a more optimized version is implemented based on the work of Vasily et. al. [77], which can reach 187 GFLOPS. In comparison, the CUDA SDK version has a throughput of 81
GFLOPS. In this section, we use matrix multiplication as an example to illustrate our compilation process.

The naïve kernel, i.e., the input to our compiler, is shown in Figure 3.2a. In the kernel, there are two input arrays, \( a \) and \( b \), from the global memory. The compiler converts the accesses to array \( a \) into coalesced ones, as shown in Figure 3.3a. Based on detected data sharing, the compiler determines that neighboring thread blocks along the X direction can be merged to improve reuse of array \( a \) and neighboring thread blocks along the Y direction can be merged to improve memory reuse of array \( b \). As the access to array \( a \) is a R2S (read-to-shared memory), the compiler chooses to perform thread-block merge. As the access to array \( b \) is R2R (read-to-register), the compiler chooses thread merge as discussed in Section 3.3.5. The next question is then how many thread blocks should be merged along either direction? As discussed in Section 3.4, the heuristic is to put at least 128 threads in each thread block and to generate different versions of kernel functions depending on the number of threads/thread blocks to be merged. Figure 3.10 shows the performance effect on GTX 280 of the number of merged threads/thread blocks in either direction. It can be seen that the optimal performance for different sizes of input matrices is achieved with merging 16 thread blocks along the X direction and 16 threads along the Y direction.

![Speedups over naïve kernels](image)

**Figure 3.11.** The speedups of the optimized kernels over the naive ones. (the input to reduction is a vector of 16M floats).
3.6 Experiments

3.6.1 Experimental Methodology

We implemented the proposed compiler framework in Cetus, a source-to-source compiler infrastructure for C programs [44]. The CUDA language support in Cetus is ported from MCUDA [74]. The compiler optimizes the naïve kernel functions of the algorithms listed in Table 3.1, all of which compute a single element at the position (idx, idy). The numbers of lines of code (LOC) of these naïve kernel functions are included in Table 3.1 to illustrate their programming complexity/simplicity. Among the kernels, ‘#pragma’ is used in the reduction kernel to convey the information of input vector length and the actual output to the compiler. The output of our compiler, i.e., the optimized kernel, is compiled by the CUDA compiler, nvcc, to generate the GPU executable file. In our experiments, we used both NVIDIA GTX8800 and NVIDIA GTX280 GPUs with CUDA SDK 2.2 and a 32-bit CentOS 5.2 operating system. Our compiler code, the naïve kernels, and the optimized kernels are available at [31]. The implementation detail of the compiler is introduced in [88].

Table 3.1. A list of the algorithms optimized with our compiler.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>The size of input matrices/vectors</th>
<th>Num. of LOC in the naïve kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>transpose matrix vector multiplication (tmv)</td>
<td>1kx1k to 4kx4k (1k to 4k vec.)</td>
<td>11</td>
</tr>
<tr>
<td>matrix mul. (mm)</td>
<td>1kx1k to 4kx4k</td>
<td>10</td>
</tr>
<tr>
<td>matrix-vector mul. (mv)</td>
<td>1kx1k to 4kx4k</td>
<td>11</td>
</tr>
<tr>
<td>vector-vector mul. (vv)</td>
<td>1k to 4k</td>
<td>3</td>
</tr>
<tr>
<td>reduction (rd)</td>
<td>1-16 million</td>
<td>9</td>
</tr>
<tr>
<td>matrix equation solver (strsm)</td>
<td>1kx1k to 4kx4k</td>
<td>18</td>
</tr>
<tr>
<td>convolution (conv)</td>
<td>4kx4k image, 32x32 kernel</td>
<td>12</td>
</tr>
<tr>
<td>matrix transpose (tp)</td>
<td>1kx1k to 8kx8k</td>
<td>11</td>
</tr>
<tr>
<td>Reconstruct image (demosaicing)</td>
<td>1kx1k to 4kx4k</td>
<td>27</td>
</tr>
<tr>
<td>find the regional maxima (imregionmax)</td>
<td>1kx1k to 4kx4k</td>
<td>26</td>
</tr>
</tbody>
</table>
### 3.6.2 Experimental Results

In our first experiment, we examine the effectiveness of our compiler optimizations. Figure 3.11 shows the kernel speedups of the optimized kernels over the naïve ones running on both GTX8800 and GTX 280 GPUs. From the figure, it can be seen that the compiler significantly improves the performance using the proposed optimizations (15.1 times and 7.9 times on average using the geometric mean).

To better understand the achieved performance, we dissect the effect of each step of our compilation process and the results are shown in Figure 3.12. The performance improvement achieved in each step is an average of all applications using the geometric mean. Since data vectorization is designed to handle complex numbers and all the inputs in this experiment are scalar numbers, this step has no effect. From Figure 3.12, it can be seen that thread/thread-block merge has the largest impact on performance, which is expected as tiling and unrolling achieved with this optimization is critical for locality and parallelism optimizations. Between the two GPUs, GTX280 benefits less from the optimizations due to its improved baseline performance (i.e., naïve kernels) as the hardware features more cores and higher memory bandwidth. Prefetching shows little impact in our results. The reason is that after thread/thread-block merge, the kernel consumes many registers. When allocating registers for prefetch, either the degree of thread merge must be reduced or the off-chip local memory may have to be used, resulting in degraded performance. Therefore, when registers are used up before prefetching, the prefetching step is skipped by our compiler. Elimination of partition camping shows larger impact on GTX280 than GTX8800. One reason is due to the input data sizes used in our experiments. For example, there is significant partition camping on GTX280 when transposing a 4kx4k matrix as it has 8 partitions and the partition size is 256 bytes. For the same input on GTX8800 which has 6 partitions, the accesses become more evenly distributed and eliminating partition camping has little effect. When transposing a 3kx3k matrix on GTX8800, however, eliminating partition camping results in a 21.5% performance improvement.
Among the algorithms in Table 3.1, six are implemented in the CUDA CUBLAS library. In the next experiment, we compare our optimized kernel with the highly tuned CUBLAS v2.2 on GTX 280. Figure 3.13 shows the performance comparison of the algorithms with different input sizes. From Figure 3.13, we can see that the kernel optimized by our compiler achieves consistently better performance than CUBLAS 2.2 for transpose matrix vector multiplication (tmv), matrix vector multiplication (mv), vector vector multiplication (vv), and matrix equation solver (strsm) for different
input sizes. For matrix multiplication (mm) and reduction (rd), the performance of our optimized code is very close to CUBLAS 2.2 (within 2% difference). On average (based on the geometric mean), our performance improvement over CUBLAS varies from 26% to 33% for different input sizes.

To study the effect of data vectorization, we chose the reduction (rd) algorithm since rd is the only algorithm in our study that has a corresponding version for complex numbers (CublasScasum) in CUBLAS. We changed the naïve kernel of rd to process complex numbers by using two float-type variables to read the real (A[2*idx]) and imaginary (A[2*idx+1]) parts of a complex number instead of a single float2 variable. Then, we optimized this naïve kernel with and without the data vectorization step. For different input sizes, we compared the performance of the two optimized kernels (labeled ‘optimized_wo_vec’ and ‘optimized’, respectively) and the results are show in Figure 3.14.

Figure 3.14. The effect of data vectorization on reduction with complex number inputs.
Figure 3.15. Performance comparison between CUDA SDK and our optimized kernel on matrix transpose.

From Figure 3.14, we can see that data vectorization significantly improves the performance. One reason is the improved memory bandwidth due to the use of float2 data types as discussed in Section 3.2. Another reason is the side effect of memory coalescing. Without data vectorization, the compiler recognized that the array accesses to both real and imaginary parts (A[2*idx] and A[2*idx+1]) are not coalesced. So, it uses shared memory as temporary storage to generate coalesced memory accesses as discussed in Section 3.3.3. In comparison, the accesses in the kernel after data vectorization, A[idx], is coalesced. As a result, the data are directly loaded into registers for computation. Although the compiler uses the shared memory to improve memory reuse for both vectorized and un-vectorized versions, there are more shared memory accesses in the un-vectorized kernel ‘optimized_wo_vec’ due to code transformation for coalescing. These extra shared memory accesses contribute to the performance differences between the ‘optimized_wo_vec’ and ‘optimized’ kernels.

Among all the kernels, transpose (tp) and matrix-vector multiplications (mv) exhibit the partition camping problem. Ruetsch and Micikevicius [67] proposed diagonal block reordering to address the issue with transpose and their implementation is included in the latest CUDA SDK. In Figure 3.15, we compare the performance of our optimized kernel (labeled ‘optimized’) with theirs (labeled ‘SDK new’) and we also include the previous CUDA SDK version for reference (labeled ‘SDK prev’). Since tp does not have any floating point operations, the effective bandwidth is used. From Figure 3.15, it can be seen that although our compiler uses the same approach to eliminate partition camping, the remaining optimizations taken by our compiler result in better performance than the version in the latest SDK.
In mv, the thread blocks are in one dimension. Therefore, diagonal block reordering cannot be applied. Our compiler uses the address offset approach described in Section 3.3.7 and the results are shown in Figure 3.16. It can be seen that for different input sizes, even without partition camping elimination, our optimized kernel (labeled ‘Opti_PC’) already achieves better performance than CUBLAS and eliminating partition camping (labeled ‘optimized’) further improves the performance.

In summary, our experimental results show that our optimizing compiler generates very high quality code and often achieves superior performance even compared to the manually optimized code in CUDA CUBLAS and SDK.

![Figure 3.16. Performance of mv using the naïve kernel, the optimized kernel without partition camping elimination (labeled ‘Opti_PC’), the optimized kernel, and CUBLAS.](image)

### 3.7 Limitations

Although the proposed compiler can dramatically improve the performance over naïve kernel functions, the fundamental limitation is that it cannot change the algorithm structure. Instead, our compiler can be used to facilitate algorithm-level exploration. The reasons are two-fold. First, developers can leverage our aggressive compiler optimizations so that they do not need to optimize their implementations of each candidate algorithm. Secondly, the relatively good understandability of our optimized code may give a hint of what algorithms are better suited. Taking 1D fast Fourier transform (FFT) as an example, when the naïve kernel (50 lines of code) simply uses 2-point FFT in
each step of the Cooley–Tukey algorithm [19], the throughput is 24 GLOPS for computing the FFT of $2^{20}$ complex numbers on GTX280. Our compiler optimizes the naïve kernel by merging threads and the resulting kernel computes 8-point FFT in each step, which delivers a throughput of 41 GFLOPS, significantly better than CUFFT 2.2 (26GFLOPS). The compiler generated 8-point FFT version, however, is not as good as a naïve implementation of 8-point FFT (113 lines of code with a throughput of 44 GFLOPS). The reason is that the compiler generated version uses multiple 2-point FFT calculations for an 8-point FFT. On the other hand, as our compiler generated code is reasonably understandable, it serves as a good guideline for algorithm exploration: changing the naïve kernel from 2-point FFT to 8-point FFT, for which the compiler can further optimize the performance to achieve 59 GFLOPS. More elaborate algorithm-level development by Govindaraju et. al. [25] as well as the one used in CUFFT2.3 achieves even higher performance (89 GFLOPS), indicating that our compiler facilitates but cannot replace intelligent algorithm-level exploration.

3.8 Related Work

CUDA [56] provides a relatively simple programming model to application developers. However, many hardware details are exposed since it is critical to utilize the hardware resources efficiently in order to achieve high performance. Given the non-linear optimization space, optimizing GPGPU programs has been shown to be highly challenging [65]. To relieve this task from developers, there has been some recent work on compiler support for GPGPU optimization. Ryoo et. al. [66] defined performance metrics to prune the optimization spaces. G-ADAPT [43] is a compiler framework to search and predict the best configuration for different input sizes for GPGPU programs. Compared to our proposed approach, this compiler takes the optimized code and aims to adapt the code to different input sizes, while ours optimizes the naïve kernel functions.

One closely related work to ours is the optimizing compiler framework for affine loops by Baskaran et. al.[10][11][12]. Their compiler uses a polyhedral model to empirically search for best loop transformation parameters, including the loop tiling sizes and unrolling factors. It is reported that their compiler achieves similar performance to CUBLAS1.0 for matrix multiplication and better performance for other kernels. In comparison, our proposed compiler also uses empirical search to determine the best parameters to merge threads/thread blocks. The difference is that we propose a novel way to achieve the effect of loop tiling and loop unrolling. In our proposed approach, we start from the finest-grain work item and aggregate work items together to exploit data reuse through
registers and share memory. This approach fits particularly well with GPGPU programming models where work items are typically defined in a 2D/3D grid and aggregating work items usually bears a clear physical meaning in terms of the workload of each thread and each thread block. In addition, we propose explicit rules to check memory coalescing and approaches to covert non-coalesced accesses into coalesced ones. For the applications that we studied, including matrix multiplication, our compiler achieves much better performance (superior or close to CUBLAS 2.2, which is significantly improved over CUBLAS 1.0). In addition, the loop transformed code generated based on polyhedral models is often quite complex [61] while our optimized code has relatively good understandability.

Our compiler shares a common goal with CUDA-lite [76]: the user provides a kernel function which only uses the global memory and the compiler optimizes its memory usage. In CUDA lite, the compiler uses the programmer provided annotation to improve memory coalescing. It also performs loop tiling to utilize shared memory. In comparison, our compiler does not require user annotation. More importantly, our compiler does not only improve memory coalescing but also effectively achieves data sharing with the proposed thread/thread-block merge techniques. In addition, our compiler distinguishes memory reads based on their target, the register or the shared memory, to make best use of either type of resource for data reuse.

One interesting way to automatically generate GPGPU programs is to translate OpenMP programs to CUDA programs [45]. Our proposed compiler is complementary to this work as it can be used to further optimize the CUDA kernel functions generated from OpenMP programs.

### 3.9 Conclusions

In this chapter, we present a compiler framework to optimize GPGPU programs. A set of novel compiler techniques is proposed to improve GPU memory usage and distribute workload in threads and thread blocks. Our experimental results show that the optimized code achieves very high performance, often superior to manually optimized programs.

In our future work, we plan to extend our compiler to support OpenCL programs so that a single naïve kernel can be optimized for different GPUs from both NVIDIA and AMD/ATI. We are also investigating detailed analytical performance models to simply the effort in design space exploration.
Chapter 4

Apricot: An Optimizing Compiler and Productivity Tool for x86-compatible Many-core Coprocessors

4.1 Introduction

Intel Many Integrated Core Architecture (MIC) has been inspired by the Intel Larrabee GPGPU processor Error! Reference source not found. that is widely considered a hybrid between a GPU and a multi-core CPU. It can be summarized as an x86-based many-core architecture that is intended for use as a coprocessor in much the same way as GPUs are used for general purpose computing in conjunction with CPUs. Unlike GPUs, which require significant programming effort, MIC is designed to leverage the x86 experience and benefit from existing parallelization tools and programming models (e.g., OpenMP [59], OpenCL [58], Intel Cilk Plus [14]), with the help of a few additional keywords that constitute the MIC programming model called LEO (Language Extensions for Offload).

Intel MIC and the LEO programming model represent the next step in the evolution of high performance computing using heterogeneous processors. The programming model (described in Section 4.2) is simple and general enough to be used with any x86-based coprocessor architecture with a suitable runtime implementation. In the most common usage model, highly parallel code regions in an application are offloaded to the MIC coprocessor over PCIe. The developer identifies and marks such code regions using #pragma offload and specifies data transfers between the host and coprocessor using in/out/inout clauses (as shown in Figure 4.2). The goal is to improve overall application performance by taking advantage of the large number of cores on MIC (for multi-
threading) and the wide SIMD units (for vectorization). Intel offload compiler and runtime manage the actual code and data transfer.

```
#pragma offload target(mic) in(B,C:length(n)) out(A:length(n))
{
#pragma omp parallel for shared(A,B,C) private(i)
for (i=0; i<n; i++){  
    A[i] = B[i] * C[i];
}
}
```

**Figure 4.1: MIC Software Stack and Execution Model**

**Figure 4.2: Pragma Offload Example**
Although MIC is a big step forward in terms of programmability as compared to GPUs, some amount of developer effort is required in order to get performance gains with MIC. First, due to the communication overheads between the host and the coprocessor, it is important to ensure that the code regions for offload are carefully selected, so that the speedup obtained by running them on MIC is not offset by offload overheads. Not only does this require extra effort on the part of the developer at compile time, it also poses a performance risk due to the unavailability of runtime information (such as input data sizes) crucial to assessing communication overheads at compile time. Secondly, specifying the data transfers between the host and coprocessor requires effort; moreso when minimizing data communication overhead is important. Finally, once the code and data transfers have been specified, it is necessary to optimize the application in order to obtain good performance on MIC.

The solution proposed in this chapter aims to maximize programmer productivity and application performance by partially automating the three steps. We present Apricot, an optimizing compiler for x86-based many-core coprocessors (such as Intel MIC) that minimizes developer effort by (i) automatically inserting conditional #pragma offload to mark code regions for offload, (ii) automatically inserting the corresponding in/out/inout clauses for data transfer for the marked code regions, (iii) selectively offloading some of the code regions to the coprocessor at runtime based on a cost-benefit analysis model that we have developed, (iv) applying a set of optimizations for minimizing the data communication overhead and improving overall performance. Apricot takes a best-effort approach for populating in/out/inout clauses. When the analysis fails to determine if a certain variable should be transferred to the coprocessor, it prints out a comment next to the pragma offload asking the developer to handle it themselves.

Unlike multi-core, where identifying a parallelizable loop with sufficient amount of work is generally enough to guarantee performance, a lot more effort is needed to guarantee performance for many-core coprocessors. Apricot is intended to assist programmers in porting existing multi-core applications and writing new ones to take advantage of many-core coprocessors, while maximizing overall performance. Experiments with SpecOMP and NAS Parallel benchmarks show that Apricot can successfully transform OpenMP applications to run on the MIC coprocessor with performance gains.
4.2 Background

The first MIC prototype, codenamed Knights Ferry, is built at 45 nm and consists of 32 in order x86 cores running at 1.2GHz with four threads per core, thus allowing 128 threads to run in parallel. Each core has a 32KB L1 instruction cache, 32KB L1 data cache and 256KB L2 cache. The L2 caches are kept coherent by a bi-directional ring network, effectively creating a shared 8MB L2 cache. The ring network also connects to the four on-die memory controllers. Knights Ferry is implemented on a PCIe card. A total of 2GB on-chip memory is present on the Knights Ferry board. There is no shared memory in hardware between Knights Ferry and the host processor. Each core has a newly designed SIMD vector unit consisting of 32 512-bit wide vector registers. The first MIC-based product, codenamed Knights Corner, would be built at 22nm and consist of more than 60 cores. It would have more memory, faster data transfer rates over PCIe, larger L2 cache per core and better single-thread performance as compared to Knights Ferry.

MIC coprocessor runs its own Linux-based Operating System that manages MIC-side resources. A device driver is installed on the host processor that is responsible for managing device initialization and communication over PCIe. A set of runtime libraries provide buffer management and communication capabilities between the CPU and MIC. The library invocations are auto-generated by the Intel Compiler for an application, thereby hiding the architectural complexity from the developer. Figure 4.1 shows the overview of the MIC software stack.

4.2.1 Programming model

In order to take advantage of the MIC coprocessor, the developer must identify regions of code in the application that can benefit from executing on MIC. A straightforward example would be a highly parallel hot loop. The developer can mark such regions using #pragma offload (as shown in Figure 4.2). The corresponding data/variables to be copied and direction of copy can be specified using in/out/inout clauses. The size of the data/variable to be copied can be specified using the length clause. Execution on the host processor is suspended when such a marked code region is encountered, continued on the MIC coprocessor and then resumed on the host processor after the code region on MIC has executed to completion. The actual transfer of code and data over PCIe is managed by the runtime libraries supplied with the MIC software stack, as noted earlier. The Intel Compiler auto-generates invocations to the libraries for code and data transfer for marked regions. All the marked code regions are transferred to MIC over PCIe in one shot at runtime when the first pragma offload is
encountered. However, data transfers for every marked code region, as specified through the
in/out/inout clauses, are managed individually as and when they are encountered, thus creating
significant communication overhead.

Table 4.1. Main LEO keywords– Programming Language Extensions for MIC

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma offload &lt;clauses&gt; target(mic)</td>
<td>Execute next code block on coprocessor</td>
</tr>
<tr>
<td>in &lt;var-list&gt;:&lt;length(size)&gt;</td>
<td>Specify MIC coprocessor as the target for offload</td>
</tr>
<tr>
<td>out &lt;var-list&gt;:&lt;length(size)&gt;</td>
<td>Copy from host to coprocessor</td>
</tr>
<tr>
<td>inout &lt;var-list&gt;:&lt;length(size)&gt;</td>
<td>Copy from coprocessor to host</td>
</tr>
<tr>
<td>nocopy &lt;var-list&gt;:&lt;length(size)&gt;</td>
<td>Copy from host to coprocessor and back</td>
</tr>
<tr>
<td>length &lt;size&gt;</td>
<td>Prohibit copying from host to coprocessor</td>
</tr>
<tr>
<td>alloc_if &lt;condition&gt;</td>
<td>Specify the size of the pointer data to be copied</td>
</tr>
<tr>
<td>free_if &lt;condition&gt;</td>
<td>Allocate memory on coprocessor only if condition is true</td>
</tr>
<tr>
<td><em>attribute</em>_((target(mic)))</td>
<td>Free memory on coprocessor for given pointer only if condition is true</td>
</tr>
<tr>
<td>_Shared &lt;variable/function&gt;</td>
<td>Allocate variable or function on both host and coprocessor</td>
</tr>
<tr>
<td>_Offload &lt;function&gt;</td>
<td>Fortran equivalent of offload pragma</td>
</tr>
<tr>
<td>!dir$ offload &lt;clauses&gt;</td>
<td>Variable/function is visible on both host and coprocessor</td>
</tr>
</tbody>
</table>

As noted above, the programming model for Intel MIC (i.e., LEO) includes a very small number of
programming language extensions/keywords to enable code and data transfer between CPU and MIC.
The main LEO keywords are shown in Table 4.1. The Fortran equivalent of #pragma offload is !dir$ omp offload. In addition to explicit data/memory transfer using in/out/inout clauses, implicit memory
transfers can be specified using the _Shared construct. A variable marked as _Shared can be used by
both the host and coprocessor code. This is accomplished by reserving a section of memory at the
same virtual address on both the host and coprocessor, which is automatically synchronized by the
runtime on entry to or exit from an offload call. All variables marked as _Shared are allocated to this
memory section. The implicit copy model simplifies programming, but has a number of limitations.
First, it only works with the _Offload construct that is used for marking an entire function for
offloading, which severely limits applicability. Secondly, the argument evaluation is consistent with
Cilk, in that the arguments are always evaluated on the host-side even if marked _Shared. Thirdly, it has poor performance as compared to the explicit copy model, due to the overheads of redundant synchronization. Fourthly, in practice it would only be helpful when there is ample memory on the coprocessor side to accommodate the longer lifetime of _Shared variables. In this chapter, we focus on the mainstream programming model that uses pragma offload and in/out/inout clauses.

OpenACC [57] is an emerging standard for accelerometers and coprocessors. It bears strong resemblance to the LEO programming model. Apricot, with some front-end modifications, can potentially be used to port and optimize OpenACC applications as well.

![Figure 4.3: Apricot Design Overview](image-url)
4.3 Apricot Overview

The primary input to Apricot is a C/C++/Fortran application, where the parallelizable loops have been annotated with OpenMP or CilkPlus constructs. The output is an optimized application that runs on the (multi-core processor + MIC coprocessor) and yields better performance. The modifications needed for porting the application to MIC and the performance optimizations are applied by Apricot, as described below.

For each parallelizable code section (marked with OpenMP or CilkPlus), Apricot performs liveness analysis to determine the variables that need to be copied in to MIC (live-in variables) and copied out of MIC (live-out variables), if the code section were to be offloaded. This would be needed for populating the in/out/inout clauses used for specifying data transfers (described in Section 4.4). It then performs array bound analysis to determine the start and end location of each array/pointer used in the code section. This is done for the purpose of populating the length clause that specifies the number of bytes to be copied starting from a particular memory location (described in Section 4.4.1).

In the second phase, Apricot inserts #pragma offload right before the offloadable code section, along with the in/out/inout and length clauses. The code section is guarded by an if condition that invokes our cost-benefit analysis function (at runtime) to determine if offloading this particular code section to the coprocessor would be beneficial. The original unmodified code section is placed in the else branch and gets executed on the host processor if the if condition returns false. Figure 4.6 shows the transformed code. We call this transformation conditional offload insertion (described in Section 4.4.2). The code in the then branch is further optimized for performance. Conditional offload insertion is carried out for every parallelizable code section, and obviates the developer effort required for (i) experimentally identifying the code sections that would benefit from Intel MIC, (ii) inserting pragma offload and determining the parameters of the in/out/inout and length clauses.

In the third phase, Apricot applies a number of performance optimizations to the code in the then branch, namely: (i) Offload Hoisting– pragma offload is hoisted out of the parent loop recursively when legal and profitable, (ii) Data Declaration Optimization– certain data declarations are moved inside the offload code section to reduce data transfer overhead, (iii) Data Reuse Optimization– if two successive offload code sections share data, the data transfers are combined, and (iv) Malloc-to-memalign Conversion– DMA transfers are triggered whenever possible. These optimizations are
crucial to performance; we describe them in Section 4.4.3. Please note that the traditional compiler optimizations (including vectorization) are performed by the Intel Compiler independent of our optimizations.

In order to decide if a code section should be offloaded at runtime, the cost-model function would need information about the amount of work being done in the code section and the amount of data being transferred in to and out of it. Apricot has a fourth phase that inspects every offload code section after all the transformations have been applied, determines the amount of computation being performed, the amount of data transfer needed and populates the parameter list for the cost-benefit analysis function (as described in Section 4.4.4). The cost-benefit analysis function invokes a set of heuristics to determine if it would be profitable to offload the code section to MIC, as described in Section 4.4.4.2.

In addition to taking OpenMP applications as input, Apricot also accepts applications that have already been modified for MIC, and optimizes them for performance. We introduce a construct called #pragma conditional-offload that extends the semantics of #pragma offload (Section 4.4.2.2). The semantics of #pragma conditional-offload state that a code block will be offloaded only if the cost-model finds it profitable to do so at runtime. This allows the developer to freely put a pragma conditional-offload before any parallelizable section without worrying about performance consequences and let the compiler and runtime decide which of them should be offloaded. The code is optimized for performance by applying the optimizations mentioned above.

Finally, if the input application is a C/C++ application with no OpenMP/CilkPlus constructs, Apricot invokes a module for identifying parallelizable code sections. In the current design, this translates to invoking the Intel Compiler’s auto-parallelizer [6], which identifies some of the parallelizable code sections. Once the parallelizable code sections are identified, Apricot inserts #pragma omp parallel for every such code section and then goes on to apply conditional offload insertion and performance optimizations in the same way as for an OpenMP application. This approach broadens the scope of Apricot. The component for identifying parallelizable code sections can be developed over time.

Figure 4.3 shows the overall design of Apricot. The primary input to Apricot is (i) OpenMP/CilkPlus applications written for multi-core that get transformed and optimized to take advantage of MIC, and (ii) unoptimized MIC applications that get enhanced with our cost-model and optimized for better
performance. In addition, Apricot has limited handling for vanilla C/C++ applications, which are first annotated with OpenMP pragmas and then handled the same way as OpenMP applications.

### 4.4 Apricot Design

Apricot is implemented on top of gcc [26] as a source-to-source compiler, and intended for use as a pre-processing tool by developers. In this section, we present the design of the different phases and components of Apricot.

#### 4.4.1 Phase I

We add support for parsing pragma offload, cilk_for, !omp parallel do and pragma conditional-offload to gcc fronted. In order to simplify analysis, we add support for ignoring compiler generated variables that are introduced in the process of lowering OpenMP pragmas. For every offloadable code

```c
void f()
{
  ...
  // candidate code section
  #pragma omp parallel for ..
  for(i=0; i < 100; i++)
  {
    int x = a;
    A[i] = B[i];
    C[i] = D[i];
    D[i] = x*i;
  }
  ...
  E[k] = A[k]*x;
  F[k] = D[k]*k;
  ...
}
LIVE_IN = {a, B, D}, LIVE_OUT = {x, A, D}
IN = {a, B}, OUT = {x, A}, INOUT = {D}
State of pragma offload after liveness analysis (length clauses are populated afterwards); note that array A is added to the inout clause:
#pragma offload target(MIC) in(a) in(B:length(..)) out(x) inout(A:length(.,),D:length(.,))
```

**Figure 4.4: Example of Liveness Analysis**
section marked with OpenMP/CilkPlus, liveness analysis is carried out followed by array bound analysis. The results are stored in a data structure called offload_regions created during parsing. In order to trigger this phase, the developer has to specify -gencodeMIC flag during compilation. This flag tells Apricot that the input is an OpenMP/CilkPlus application that needs to be analyzed and transformed to generate code that can execute on MIC.

### 4.4.1.1 Liveness Analysis

Liveness analysis [37] is performed before creation of SSA [3], and handles simple arrays and pointers in addition to scalars. Prior to performing liveness analysis, all function invocations present inside the code section are eliminated by inlining the function bodies. If inlining is not possible, the code section is marked as not-offloadable (which is a rare occurrence). We leverage the existing gcc modules for live-in/live-out analysis for scalars, tailor them for our purpose and extend them to handle pointers and arrays. This is done with the help of the memory tag infrastructure in gcc, which can be used for tracking reads and writes to pointers. There is no special representation for arrays in the gcc IR, so for the purpose of liveness analysis, they are handled the same way as pointers: a write to a memory location serves as a def for the pointer/array that points to that memory location. Due to the complexity of handling pointers and arrays, we take a best effort approach. Indirect memory references, nested arrays and structs are not handled in the current implementation, due to the limitations of the array bound analysis pass (described below).

Bitmaps are used to record LIVE_IN and LIVE_OUT sets for each basic block. At the end of the analysis, the LIVE_IN set for the first basic block of the offload code section is stored as the IN set for the code section. The LIVE_OUT set of global variables is computed a little differently. Since our inter-procedural analysis is limited in its capability, we cannot guarantee the absence of the use of a global variable in the rest of the application. To err on the conservative side, the LIVE_OUT set includes all global variables that are defined/re-defined inside the code section. The LIVE_OUT set of the last basic block of the offload code section is stored as the OUT set for the code section.

The variables that are common to both the IN and OUT sets are used for populating the inout clause, which signifies the set of variables that should be copied into MIC prior to the execution of the code section and copied out of MIC after completion. The variables that are present only in the IN set are used for populating the in clause, which signifies the set of variables that should be copied into MIC. The scalars present only in the OUT set populate the out clause, which signifies the set of variables
that should be copied out of MIC after the code section executes to completion. Note that all the pointers/arrays that are only present in the OUT set are added to the inout clause (instead of the out clause as expected). The reason for this is as follows: when an array is copied back to host memory, any unassigned memory locations (on the MIC side) will cause null/garbage to be written onto corresponding memory locations on the host side. To prevent this from happening, it is important to copy the contents of the array into MIC memory, even if there are no reads to it. Figure 4.4 shows a simple example of liveness analysis in Apricot.

4.4.1.2 Array Bound Analysis

For any non-scalar variable, i.e. arrays and pointers, the size of the memory to be copied is specified using the length clause, as shown in Figure 4.6. Estimating the section of memory that would be used inside a code section, for a given pointer, is non-trivial. A simple approach is to trace the declaration of the array/pointer and pass the entire size of the array/pointer (when available) to the length clause. This, however, would result in redundant data transfers. We attempt to solve this problem by estimating the section of memory being accessed inside the offload code section (for a given pointer), by using the scalar evolution [60] representation in gcc IR (which bears resemblance with the polyhedral model [15]), and information about loop upper/lower bounds. This approach is similar to array section analysis [20][21][68], commonly used for array privatization [42].

```
for(i=30; i < 100; i++) {
    x = A[i];
}
```

Simplified IR:
```
a = 30;
loop_1
    b = phi(a,c)
    x = A[b];
    c = b + 1;
    if(c > 100) exit_loop
endloop
```

Scalar Evolution Rep. for b: \{30,+,1\} \Rightarrow initial value=30, increment=+1

Upper Bound of loop: 100
Bounds of array A: \{30,100\}

**Figure 4.5: Simple Example of Array Bound Analysis**
We leverage existing gcc modules for this purpose. As mentioned before, arrays and pointers are handled in the same way. The basic idea is to identify the memory access pattern of a pointer by parsing it as a function of the loop indices (using scalar evolution representation). For each loop index, the corresponding stride, lower bound and upper bound are extracted. This allows us to get an estimate of the bounds of the pointer/array. Figure 4.5 shows a simple example.

Discussion: Indirect memory references, structs and nested arrays are not handled in the current implementation. This will be gradually improved over time as we beef up our liveness and array bound analyses. In order to keep the coverage of Apricot high, we conform to a best effort semi-automatic approach. Since Apricot outputs transformed source code, the developer can make further changes as needed. If Apricot is not able to estimate the liveness or size of a given pointer for an offload code section (such as a struct or a nested array), it prints out a comment next to the pragma offload clause asking the developer to handle it themselves. In other words, Apricot populates the in/out/inout and length clauses to the best of its ability and asks the developer to add information about the missing ones (if needed).

### 4.4.2 Phase II

The first round of code transformations are applied in phase II. As mentioned earlier, when -gencodeMIC flag is used, the input is assumed to be an OpenMP/CilkPlus application. Apricot carries out conditional offload insertion for every offload code region parsed in phase I. If –optimizeMIC flag is used, the input is assumed to be a MIC application. In this case, Apricot lowers all the #pragma conditional-offload statements in the application, which is similar to conditional offload insertion with a few minor differences.
4.4.2.1 Conditional Offload Insertion

Conditional offload insertion consists of three steps. In the first step, an if-then-else block is created, the original code section is placed in the else branch and a copy of the code section is placed in the then branch. An invocation to the cost-benefit analysis function (cbf()) is introduced as the predicate. The parameter list for cbf() is empty at this point. In the second step, #pragma offload is inserted at the beginning of the then branch along with empty in/out/inout and length clauses. In the third step, the information stored in offload_regions created during phase I is used to populate in/out/inout and length clauses. Figure 4.6 shows the transformation for a simple code snippet.

4.4.2.2 Lowering #pragma conditional-offload

This is very similar to conditional offload insertion. An if-then-else block is created, the original code section is placed in the else branch and a copy of the code section is placed in the then branch. An invocation to the cost-benefit analysis function (cbf()) is introduced as the predicate. The #pragma
conditional-offload is replaced by #pragma offload in the then branch, and deleted from the else branch. An entry is made into offload_regions for this code section. The transformation is shown in Figure 4.7.

```c
#pragma conditional-offload target(mic) in(B,C:length(n))
   out(A:length(n))
{
   # pragma omp parallel for shared(A,B,C) private(i)
   for (i=0; i<n; i++)
   A[i] = B[i] * C[i];
}
}

(a) Original Code
```

```c
if(cbf ()){
   #pragma offload target(mic) in(B,C: length(n))
   out(A: length(n))
   {
   # pragma omp parallel for shared(A,B,C) private(i)
   for (i=0; i<n; i++)
   A[i] = B[i] * C[i];
   }
   }
else{
   # pragma omp parallel for shared(A,B,C) private(i)
   for (i=0; i<n; i++)
   A[i] = B[i] * C[i];
   }
}

(b) Transformed Code
```

**Figure 4.7: Lowering #pragma conditional-offload**

### 4.4.3 Phase III

The code generated from phase II is optimized for performance in this phase. We outline a few prominent optimizations that have been experimented with so far. These optimizations are targeted
towards minimizing the communication overheads between the host processor and MIC. Traditional compiler optimizations (e.g., vectorization) are performed by the Intel Compiler independently.

```
for(j=0; j<100; j++){
    #pragma offload target(mic) in(B,C: length(n)) out(A: length(n))
    {
        # pragma omp parallel for shared(A,B,C) private(i)
        for (i=0; i<n; i++){
            A[i] = B[i] * C[i];
        }
    }

    #pragma offload target(mic) in(A,B: length(n)) out(D: length(n))
    {
        # pragma omp parallel for shared(A,B,D) private(i)
        for (i=0; i<n; i++){
            D[i] = A[i] + B[i];
        }
    }
}
```

(a) Original Code

```
#pragma offload target(mic) in(A,B,C: length(n)) out(A,D:length(n))
{
    for(j=0; j<100; j++){
        #pragma omp parallel for shared(A,B,C) private(i)
        for (i=0; i<n; i++){
            A[i] = B[i] * C[i];
        }

        #pragma omp parallel for shared(A,B,D) private(i)
        for (i=0; i<n; i++){
            D[i] = A[i] + B[i];
        }
    }
}
```

(b) Transformed Code

**Figure 4.8: offload Pragma Hoisting**
4.4.3.1 Offload Hoisting

Optimal placement of pragma offload is essential for performance. Consider the code snippet in Figure 4.8(a). The offload overhead for the pragma offload as shown in Figure 4.8(a) is \(100 \times (t_1 + t_2)\), where \(t_1\) and \(t_2\) represent the time it takes to complete offload/data transfer for the two code sections respectively. Hoisting pragma offload out of the parent loop (as shown in Figure 4.8(b)), would reduce the data transfer time to \(t_1 + t_2\), significantly improving performance.

In general, this optimization aims to exploit opportunities for work consolidation by constructing one large code section for offload from several small ones, in order to minimize the communication overheads between the host processor and MIC. In the current design, this is implemented as iterative hoisting of pragma offload’s. If a code region marked by pragma offload is found to have a parent loop, analysis is carried out to determine if it would be profitable to hoist the pragma offload outside the parent loop. If so, the pragma offload is hoisted out and all the pragma offload’s inside the parent loop are deleted. Liveness and array bound analyses are invoked on-demand for the outer loop in order to populate the in/out/inout clauses.

For offload hoisting to be profitable, it is important to make sure that all the hot inner loops are parallelizable and that the serial code between them is minimal. We have interfaced the GNU gprof [27] profiler with Apricot in order to determine profitability of this optimization. Gprof is invoked from within Apricot before Phase III. The generated profile information is processed and stored on a per-loop basis for use by later optimizations. Currently, offload hoisting is the only optimization that makes use of profile information. If \(t_1\) is the time spent in the outer loop, and \(t_2\) is the total time spent in the inner parallelizable loops (marked by OpenMP pragma’s), offload hoisting is performed only when \(t_1 / t_2 > c\). The value of \(c\) is currently set to 0.8. In other words, offload hoisting is applied when at least 80% of the time spent in the parent loop is spent in parallel code regions.

4.4.3.2 Data Declaration Optimization

Data declaration optimization is a very simple but effective transformation for reducing communication overheads. For all the variables in the LIVE_IN set of an offload code section that are declared outside, we try to ascertain if the variable is used only inside the code section. This is done by looking at the use-def chain and making sure that the declaration is local to the function and that there are no uses/definitions of the variable outside the code section, except for the declaration itself. We move the declaration of such variables inside the offload code block to avoid the redundant copy-
In C/C++, it is common practice to declare a data variable at the top of a function, which creates opportunities for applying this optimization.

![Figure 4.9: Data Reuse Optimization: Transformation]

```c
#pragma offload target(mic) in(m:length(1000))
   out(a: length(1000))
{
  #pragma omp parallel for shared(a,m) private(i)
   for(i=0; i < 1000; i++)
   {a[i] = m[i];}
}
... /* no writes to m */
#pragma offload target(mic) in(m:length(1000))
   out(b: length(1000))
{
  #pragma omp parallel for shared(b,m) private(i)
   for(i=0; i < 1000; i++)
   {b[i] = m[i]*m[i];}
}
(a) Original Code

#pragma offload target(mic) in(m:length(1000) free_if(0))
   out(a: length(1000))
{
  #pragma omp parallel for shared(a,m) private(i)
   for(i=0; i < 1000; i++)
   {a[i] = m[i];}
}
... /* no writes to m */
#pragma offload target(mic) nocopy(m: length(1000) alloc_if(0))
   out(b: length(1000))
{
  #pragma omp parallel for shared(b,m) private(i)
   for(i=0; i < 1000; i++)
   {b[i] = m[i]*m[i];}
}
(b) Transformed Code
```
4.4.3.3 Malloc-to-memalign Conversion

The use of DMA can significantly improve performance. In order for the Intel Compiler to generate DMA transfers, the pointer address must be 64-byte aligned and the data size must be a multiple of 64, which can be done using posix_memalign. Malloc-to-memalign optimization traces the malloc for a given data pointer in the in/inout clause, and replaces it by a suitable posix_memalign when possible. Function mult64 is defined to round off the size of the allocated pointer to the next multiple of 64.

4.4.3.4 Data Reuse Optimization

When a data variable is copied into MIC, memory is allocated for it by default unless otherwise specified using the alloc_if clause; alloc_if allows memory to be conditionally allocated based on the truth value of the predicate. Similarly, after an offload section finishes to completion, the memory is deallocated by default, unless otherwise specified using the free_if clause, which allows memory to be conditionally deallocated.

If a data variable is shared between multiple offload code sections or between successive executions of a code section, such that there are no writes to it on the host side (in between those executions), it is generally beneficial to retain the data in MIC memory and consequently delete all redundant data transfers associated with it. This can be done with the help of alloc_if, free_if and nocopy clauses. A nocopy clause indicates that a variable’s value already exists in MIC memory and prohibits copy-in.

As mentioned before, in order to apply the Data Reuse optimization for a given data variable, we need to ensure that there are no definitions of the variable on the host side in between two occurrences (i.e., uses or definitions) on MIC side. This can be done by inspecting the use-def chain for the variable. The analysis needed for data reuse across two successive executions of an offload code section (e.g., in a loop) is simpler than the analysis for data reuse across two difference code sections. The complexity arises due to the conditional nature of offload as introduced by our compiler whether a code section will execute on the host processor or be offloaded to MIC is not known at compile time. The data reuse optimization creates a dependence between two code sections. If only one of them gets offloaded to MIC, incorrect results may be generated. In order to guarantee safety, it is important to ensure that both the code sections, for which data reuse optimization is being applied, get offloaded to MIC or neither does.
For applying data reuse optimization across two different code sections, if-fusion is performed. Since cbf() is part of Apricot, if-fusion can be applied without danger of side effects. Data reuse analysis is then applied to the two code sections in the then branch by inspecting the use-def chains for all variables in the in/out/inout clauses. The variables that get defined on the host side in between the two code sections are not considered as candidates for reuse.

If a data variable m occurs in the in/inout clause of the first pragma and the in clause of the second pragma, a free_if(0) is inserted in the first pragma and a nocopy along with alloc_if(0) is inserted in the second pragma. This informs the compiler that m is not to be deallocated after the completion of first code section and that it is not to be copied in or allocated for the second code section. The transformation is shown in Figure 4.9. If a data variable m occurs in the inout clause of both the pragmas, such that there are no uses of it in between on the host side, then m is moved from the inout to the in clause of the first pragma and from the inout to the out clause of the second pragma. A free_if(0) is inserted for it in the first pragma and a nocopy along with alloc_if(0) is inserted in the second pragma. This tells the compiler that m is to be reused between the first and second code section and that it should be copied out only after the completion of the second code section.

In order to apply data reuse analysis across successive executions of an offload code section inside a loop, the use-def chains for all variables in the in/out/inout clause are inspected. If a data variable in the in/inout clause does not get defined outside the code section within the surrounding loop, data reuse optimization is applied to the code section. An empty code section (surrounded by pragma offload) is created and placed right before the original code section. The empty code section only gets executed for the first iteration of the loop. All data variables in the in clause that do not get defined outside the code section in the surrounding loop are moved to the in clause of the empty code section. Corresponding nocopy, free_if(0) and alloc_if(0) clauses are inserted. All data variables in the inout clause that do not get defined outside the code section in the surrounding loop (but may get used) are moved to the in clause of the empty code section and the out clause of the original code section. If any of those data variables get neither defined nor used outside the code section in the surrounding loop, another empty code section is created and placed right after the loop, and all such variables are moved from the out clause of the original code section to the out clause of this empty code section. Corresponding nocopy, free_if(0) and alloc_if(0) clauses are inserted. Since the two empty code sections are only executed once, the copy-in/copy-out and memory allocation of the data variables is done once per loop as opposed to once per iteration.
4.4.4 Phase IV

In the final phase of compilation with Apricot, all the optimized offload code sections are inspected and the parameter list for the cost-benefit analysis function (cbf()) is populated for each code section.

4.4.4.1 Inspection for Cost-benefit Analysis

The main idea behind this analysis is to come up with estimates for the amount of work done and data communicated per code section and pass them as parameters to the cost model. A code section is typically a parallelizable loop and in some cases (where offload hoisting has been successfully applied) a set of parallelizable loops with limited serial code in between. This approach is inspired by the analysis carried out for auto-parallelization of loops, as in the Intel Compiler [6][32].

In order to get an estimate for the amount of work done in the code section, Apricot identifies (i) the set of cpu operations (i.e., addition, subtraction, multiplication, exponentiation, etc) in the code section and condenses them into a single number (cpu_ops) per loop based on relative weights (that are experimentally obtained) for the different operations, (ii) the set of memory operations (i.e., loads/stores) condensed into a single number (mem_ops) per loop, and (iii) the number of loop iterations (num_iter) for each loop. The estimate for data transfer size (dsize) is obtained by adding up the data sizes of all the scalars, arrays and pointers specified the in/out/inout clauses. This information is passed as a set of parameters to cbf() for each code section.

4.5 Evaluation

Apricot has been implemented on top of the gcc infrastructure. The analyses are performed with gcc IR, but the actual transformations are applied to the source code. We take advantage of the IR-to-source mapping capabilities in gcc. Output from the different analyses are stored and maintained in the offload_regions data structure and used during transformation. This approach allows the user to make further modifications to the optimized source code and have a final say in what gets executed, thereby allowing Apricot to function as a pre-processor/best-effort compiler and be used as a productivity tool as much as a performance tool. Moreover, generating optimized source code provides additional flexibility as to how, where and when the optimized source code is eventually compiled. All the different phases and optimizations presented in the chapter have been implemented and tested.
We carry out the experiments with a Quadcore Xeon E5620 server equipped with a Knights Ferry Card. Knights Ferry has been made available to a few groups for the purpose of debugging and optimizing the runtime, compiler and other libraries for MIC in order to pave the way for Knights Corner, the first Intel MIC product. Due to confidentiality reasons, we can only report the relative gains obtained with our optimizations.

Figure 4.10: Performance gains obtained with the four optimizations applied to the baseline offload version

We use the Knights Ferry card to (i) test out correctness of conditional offload insertion, and (ii) evaluate the four optimizations presented in the chapter. We have experimented with SpecOMP [73]...
benchmarks and NAS Parallel benchmarks [50], in addition to a number of micro-benchmarks. Our experiments show that Apricot can successfully transform OpenMP applications to run on MIC by applying conditional offload insertion. In the rest of the section, we present our experimental results for two SpecOMP benchmarks: 321.equake_l and 313.swim_l and two NAS parallel benchmarks: FT and MG. For all the benchmarks, code modifications are made to convert multi-dimensional arrays to one-dimensional arrays (since the current implementation of Apricot only handles one-dimensional arrays during array-bound and liveness analysis).

321.equake_l: Finite element simulation of elastic wave propagation in large valleys due to seismic events. There is a total of 12 loops marked with #pragma omp for. The compiler auto-generates conditional #pragma offload and corresponding in/out/inout/length clauses for these loops and the benchmark executes successfully on MIC. The cost model finds that only 2 of the 12 loops would benefit from offloading to MIC. This version serves as the baseline for evaluating the four optimizations.

Figure 4.10(a) shows the results with the four optimizations applied individually and together. The Y-axis represents the time of execution of the benchmark (in timeunits) such that it is proportional to the actual time of execution (omitted for confidentiality). The numbers are averaged over multiple runs and reported with num_threads set to 64. The baseline (offload version without any optimizations applied) takes 234 time units to execute. A speedup of 2.4X, 5.2X, 11.7X and 17.3X is obtained with malloc-to-memalign conversion, data declaration optimization, offload hoisting and data reuse optimization respectively. When all the optimizations are applied together, a speedup of 24.5X is achieved. The reason why performance gains from the different optimizations are not additive is because application of one optimization affects the applicability as well as performance headroom of other optimizations. For example, when offload hoisting is applied, the gains obtained by data reuse optimization diminish because less data is transferred.

313.swim_l: Weather prediction program, used for comparing performance of supercomputers. There is a total of 11 loops marked with !omp parallel do. The cost model finds that only 2 of the 11 loops would benefit from offloading to MIC. This version serves as the baseline for evaluating the four optimizations. Figure 4.10(b) shows the results with the four optimizations applied individually and together. The baseline (offload version) takes 429 time units to execute. A speedup of 2.5X, 1.33X, 3.89X and 1.45X is obtained with malloc-to-memalign conversion, data declaration optimization,
offload hoisting and data reuse optimization respectively. When all the optimizations are applied together, a total speedup of 3.91X is obtained.

**FT:** Time integration of a three-dimensional partial differential equation using FFT. There is a total of 6 loops marked with `#pragma omp for`, of which 3 are found to benefit from offloading to MIC by our cost-model. Figure 4.10(c) shows the results with the four optimizations applied. The baseline version takes 80 time units to execute. A speedup of 1.45X, 1.1X, 2.3X and 1.9X is obtained with malloc-to-memalign conversion, data declaration optimization, offload hoisting and data reuse optimization respectively. When all the optimizations are applied together, a total speedup of 2.47X is obtained.

**MG:** Multigrid solver for a 3-dimensional potential field. There is a total of 12 loops marked with `#pragma omp for`, of which 5 are found to benefit from offloading. Figure 4.10(d) shows the results with the four optimizations applied. The baseline (offload version) takes 53 time units to execute. A speedup of 8.2X and 7X is obtained with offload hoisting and data reuse optimization respectively. A 3% gain is obtained with malloc-to-memalign conversion, while data declaration optimization does not yield any performance gains for this benchmark. A total speedup of 8.75X is obtained with all the optimizations applied together.

These experimental results indicate the significant potential of the optimizations proposed in the chapter. The results are reported for 56-64 threads which yield best performance for these benchmarks (for certain benchmarks 88-96 threads yield slightly better performance). The gains may vary as the hardware evolves but communication overheads would continue to be a performance bottleneck and software techniques would be needed to alleviate the problem.

### 4.6 Related work

Donaldson et al [22] illustrate the use of Offload C++ extensions for offloading C++ code to Cell BE. This work is nearly identical (as well as contemporary) to Intel’s LEO programming model and the corresponding compiler and runtime. Apricot is complementary to both and can be used alongside Offload C++, in the same way as it is used alongside LEO. Apricot focuses on the automatic insertion of the offload constructs, optimization of the generated code and selective offloading at runtime.

General purpose programming for GPUs (GPGPU) has received a lot of attention lately. Consequently, a number of GPGPU compilers have been proposed and developed [45][12][34][21][63]. The idea behind these compilers is to reduce the programming effort and
maximize performance for GPUs. LEO and the Intel Offload Compiler are in their infancy and have significant room for improvement, as demonstrated in this chapter. Apricot shares the goal of improving programmer productivity and application performance for this new generation of many-core coprocessors, as well as the design rationale, with GPGPU compilers. However, the manner in which the ideas are formulated and applied is very different due to the difference in the programming models as well as target architectures.

Apricot builds on top of prior work in the area of static analysis [29][37], multi-core programming [14][59] and heterogeneous computing [13][21]. The idea of conditional offload is inspired by the problem of automatic partitioning of applications for CPU/coprocessor model [36]. Minimizing communication overhead [28][34] between host processor and MIC coprocessor is the main goal of the optimizations introduced in the chapter. The inspection phase for cost-benefit analysis is inspired by auto-parallelization [6]. The liveness analysis phase bears resemblance with the approach used for automatic generation of MPI messages from OpenMP [40].

The idea of conditional offload, the cost model as well as the compiler optimizations introduced in this chapter are novel. To the best of our knowledge, this is the first attempt at investigating LEO and suggesting extensions and optimizations. These optimizations can potentially also be applied to C++ Amp [17], which is a set of programming language extensions (similar to LEO) being proposed by Microsoft for GPGPU.

4.7 Conclusion and Future Work

In this chapter, we give an overview of the new Intel MIC architecture and the associated programming model for offload called LEO. We present the design of Apricot, an optimizing compiler and productivity tool for many-core coprocessors such as MIC. The key features of Apricot are: (i) automatic transformation of legacy applications to take advantage of MIC coprocessor (ii) cost model and analysis for selective offload of code regions to the coprocessor at runtime (iii) optimization of code regions for minimizing data and communication overheads. We evaluate the optimizations with respect to Knights Ferry, a MIC prototype developed by Intel as a precursor to Knights Corner and observe up to 24X speedup. Since LEO bears strong resemblance to the OpenACC standard [57], Offload C++ [22] as well as C++ Amp [17], Apricot front-end can be extended to optimize applications written with these programming models as well.
The current implementation of liveness and array bound analysis only handles simple pointers and arrays; nested arrays and structures of pointers are not handled. We plan to extend the implementation to handle more complex cases to further improve coverage. The cost model is currently tuned with respect to Knights Ferry. We need to refine and reevaluate the cost model (as well as the optimizations) with respect to Knights Corner. We also plan to design an auto-parallelizer specifically for many-core and evaluate it independently.
Chapter 5

Shared Memory Multiplexing: A Novel Way to Improve GPGPU Performance

5.1 Introduction

Modern many-core graphics processor units (GPUs) rely on thread-level parallelism (TLP) to deliver high computational throughput. To mitigate the impact of long latency memory accesses, besides TLP, software managed on-chip local memory is included in state-of-art GPUs. Such local memory, called shared memory in NVIDIA GPUs and local data share in AMD GPUs, has limited capacity. As a shared resource among threads, shared memory is one of the key factors to determine how many threads can run concurrently on a GPU.

State-of-art GPUs manage shared memory in a relatively simple manner. When a group of threads (called a thread block or workgroup) is to be dispatched, the shared memory is allocated based on the aggregate shared memory usage of all the threads in the thread block (TB). When a TB finishes execution, the allocated shared memory is released. When there is not sufficient shared memory for a thread block, the TB dispatcher is halted.

There are two major limitations to the aforementioned shared memory management. First, the allocated shared memory is reserved throughout the lifetime of a TB, even if it is only utilized during a small portion of the execution time. Secondly, when the shared memory size (e.g., 16kB) is not a multiple of the shared memory usage (e.g., 9kB) of a TB, a fraction of shared memory (e.g., 7kB) is always wasted. These two limitations reduce the number of TBs that can concurrently run on a GPU, which may impact the performance significantly as there may not be sufficient threads to hide long latencies of operations such as memory accesses.

In this chapter, we first characterize the usage of shared memory in GPGPU (general purpose computation on GPUs) applications and make an important observation that many GPGPU
applications only utilize shared memory for a small amount of time compared to the lifetime of a TB. Then, we propose novel ways to multiplex shared memory so as to enable a higher number of TBs to be executed concurrently. These schemes include three software approaches, namely VTB, VTB_pipe, CO-VTB, and one hardware solution. Our software approaches work on existing GPUs and they essentially combine two original TBs into a new TB and add if-statements to control time multiplexing of the allocated shared memory between the two original TBs. Our hardware solution incurs minor changes to existing hardware and supports dynamic shared memory allocation and de-allocation so as to enable shared memory multiplexing with very little change in GPGPU code.

Our experimental results on NVIDIA GTX285 and GTX480 GPUs show remarkable performance gains from our proposed software approaches, up to 1.95X and 1.44X on average on GTX285 and up to 2.19X and 1.70X on average on GTX480. We also evaluate our hardware proposal using the GPGPUsim simulator [9], which shows up to 2.34X and an average of 1.53X performance enhancement.

Our work makes the following contributions. (1) We characterize shared memory usage among GPGPU applications and highlight that many GPGPU applications utilize shared memory only for a limited portion (an average of 25.6%) of the execution time of a TB; (2) we propose three software approaches to time multiplex shared memory among TBs; (3) we propose a simple hardware approach to support dynamic shared memory management; and (4) we show that our proposed software- and hardware-based shared memory multiplexing approaches are highly effective and significantly improve the performance.

The remainder of the chapter is organized as follows. In Section 5.2, we present a brief background on GPGPU architecture and highlight the importance of shared memory. We characterize the shared memory usage of GPGPU applications in Section 5.3. In Section 5.4, we present our three software approaches to multiplex shared memory. Section 5.5 discusses our hardware solution. The experimental methodology is addressed in the Section 5.6 and the results are presented in Section 5.7. Related work is discussed in Section 5.8. Section 5.9 concludes this chapter.

**5.2 Background**

As mentioned in Chapter 2, on-chip shared memory is a critical resource for GPGPU applications. The shared memory provides a mechanism for threads in the same TB to communicate with each other. It also serves as a software managed cache so as to reduce the impact of long latency memory
accesses. Since each SM has limited amount of shared memory, for many GPGPU applications, the usage of shared memory of a TB determines how many TBs can run concurrently, i.e., the degree of thread level parallelism (TLP). Besides shared memory, the register usage of each thread is another critical factor to determine the number of threads that can run concurrently. In state-of-art GPUs, both shared memory and register files (RFs) are managed similarly. When a TB is to be dispatched to an SM, the TB dispatcher allocates the shared memory and registers based on the aggregate usage of all the threads in the TB. The allocated shared memory and registers are released when the TB finishes execution. When there is not sufficient resource available in either shared memory or RF in an SM, the resource is not allocated and no TB will be dispatched to the SM.

Between shared memory and RFs, current GPUs have higher capacity in RFs. For example, on NVIDIA GTX285 GPUs, each SM has 16kB shared memory and a 64kB RF. On NVIDIA GTX480 GPUs (i.e., the Fermi architecture), each SM has a 128kB RF and a 64kB hybrid storage that can be configured as a 16kB L1 cache+48kB shared memory or a 48kB L1 cache+16kB shared memory. The latest NVIDIA GPU, GTX680 (i.e., the Kepler architecture), has the same size of shared memory per SM as GTX480 and a 256kB RF. With a high number of SPs and a larger RF in each SM, the Kepler architecture is designed to host more concurrent thread blocks/threads in each SM than the Fermi architecture, thereby increasing the pressure on shared memory. On AMD HD5870 GPUs, each CU contains 32kB shared memory (called local data share) and a 256kB RF. As a result, for many GPGPU applications, shared memory presents a more critical resource to limit the number of TBs/threads to run concurrently on an SM.

5.3 Characterization of Shared Memory Usage

To understand how GPGPU applications utilize shared memory, we select and study ten benchmarks, which have shared memory variables in the source code, as shown in Table 5.1. Among the benchmarks, MC, SP, MM, CV, RD and TP are from NVIDIA SDK [52]. FFT and HG are from AMD SDK [4]. STO [2] is from the GPGPUSim infrastructure [9]. We implemented the GPU kernel for MV, which has similar performance to CUBLAS 4.0 [53]. For each benchmark, the shared memory usage of a TB as well as the number of threads in a TB is reported in Table 5.1. Also included is the number of TBs that can run concurrently in an SM with 16kB shared memory.

From Table 5.1, we can classify the benchmarks into two categories. The first category, including MV, FFT, MC, STO, SP, HG and CV, has the characteristics that the number of threads, which can
execute concurrently in a SM, is severely limited by the shared memory capacity. The second category, including MM, TP, and RD, has the characteristics that either the shared memory usage of each TB is small or each TB has a large number of threads. For the benchmarks in the second category, the shared memory is not a bottleneck for TLP. Therefore, the target of our proposed approaches is the workloads in the first category. Another interesting observation from Table 5.1 is that for all workloads in the first category, except STO, the shared memory can be severely underutilized, as shown in the last column of Table 5.1, although the limited shared memory size is the cause for limited thread-level parallelism (TLP). The reason is that when a TB is to be dispatched, all its required shared memory needs to be available. Therefore, if the remaining shared memory is not enough for a TB, it is always wasted.

Table 5.1. Benchmarks for shared memory multiplexing

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Shared memory size per TB (Bytes)</th>
<th>Threads per TB</th>
<th>Threads (TBs) per SM</th>
<th>Actual shared memory usage per SM (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix multiplication (MV)</td>
<td>4268</td>
<td>32</td>
<td>96 (3)</td>
<td>4268x3 = 12804</td>
</tr>
<tr>
<td>Fast Fourier Transform (FFT)</td>
<td>8736</td>
<td>64</td>
<td>64 (1)</td>
<td>8736x1 = 8736</td>
</tr>
<tr>
<td>MarchingCubes (MC)</td>
<td>9324</td>
<td>32</td>
<td>32 (1)</td>
<td>9324x1 = 9324</td>
</tr>
<tr>
<td>StoreGPU (STO)</td>
<td>16304</td>
<td>128</td>
<td>128 (1)</td>
<td>16304x1 = 16304</td>
</tr>
<tr>
<td>ScalarProd (SP)</td>
<td>4144</td>
<td>64</td>
<td>192 (3)</td>
<td>4144x3 = 12432</td>
</tr>
<tr>
<td>Histogram (HG)</td>
<td>8224</td>
<td>64</td>
<td>64 (1)</td>
<td>8224x1 = 8224</td>
</tr>
<tr>
<td>Convolution(CV)</td>
<td>8300</td>
<td>128</td>
<td>128 (1)</td>
<td>8300x1 = 8300</td>
</tr>
<tr>
<td>Matrix Multiplication (MM)</td>
<td>2084</td>
<td>256</td>
<td>1024 (4)</td>
<td>2084x4 = 8336</td>
</tr>
<tr>
<td>Transpose (TP)</td>
<td>4260</td>
<td>128</td>
<td>384 (3)</td>
<td>4260x3 = 12870</td>
</tr>
<tr>
<td>Reduction (RD)</td>
<td>540</td>
<td>128</td>
<td>1024 (8)</td>
<td>540x8 = 4320</td>
</tr>
</tbody>
</table>

Next, for all the benchmarks in Table 5.1, we use simulation to determine how long the shared memory is utilized in a TB (see Section 5.6 for the detailed experimental methodology). Since the
compiler may schedule the shared memory accesses to interleave with other type instructions to improve instruction-level parallelism (ILP), if we simply consider the lifetime of shared memory usage as between the first instruction writing to the shared memory and the last instruction reading from the shared memory, we may find that the shared memory is used for almost the entire lifetime of kernel execution. To isolate the usage of shared memory from other parts of the kernel code, we insert ‘__syncthreads()’ instructions before the first write/define to and after the last read/use from the shared memory. We denote a code region surrounded by our inserted ‘__syncthreads()’ as a shared memory access region. A redefine of the shared memory variables will start a new shared memory access region. Here, any define or use of shared memory variables is based on all the threads in a TB. Then, we use the accumulated execution time of all shared memory access regions as the duration for shared memory usage. In Figure 5.1, we show the ratio of the duration of shared memory usage over the overall execution time of a TB. For each benchmark, this ratio is an average across all its TBs.

![Figure 5.1. The portion of the execution time, during which the allocated shared memory is actually utilized, of a thread block for GPGPU applications.](image)

As shown in Figure 5.1, using the geometric mean (GM) for all the benchmarks, the shared memory is only used in 25.6% of the execution time, which means that TBs do not need to use the allocated shared memory during the 74.4% of their execution time. Among these benchmarks, MM and TP have high TLP and they use the shared memory often during the lifetime of a TB. RD has high TLP.
but its performance bottleneck is global memory accesses; therefore the shared memory is idle most of the time. HG spends most of the execution time on accessing data in the shared memory, thereby showing the high ratio in Figure 5.1. For the remaining benchmarks, the TLP is limited by the shared memory capacity although their allocated shared memory is only used for a very limited amount of time compared to the lifetime of a TB.

```c
loadFromGlobal();
FFT4(0);
(saveToSM(0);    //define by multiple threads in a TB
__syncthreads();
loadFromSM(0);  //use by multiple threads in a TB
FFT4(1);
__syncthreads();
(saveToSM(1);    //re)define by multiple threads in a TB
__syncthreads();
loadFromSM(1);  //use by multiple threads in a TB
....
FFT4(4);
writeToGlobal();
```

Fig. 5.2. The pseudo code of a 1k-point FFT implementation, which uses 8736-Byte shared memory per thread block and there are 64 threads per thread block.

5.4 Shared Memory Multiplexing: Software approaches

As discussed in Section 5.3, many GPGPU applications suffer from insufficient TLP due to the limited shared memory capacity. In this section, we propose three software approaches to time multiplex shared memory to boost TLP. The key idea of these three approaches is the same: we combine multiple original TBs into a larger one and introduce control flow to manage how the shared memory is accessed among the original TBs. The difference among the three approaches lies in how to overlap shared memory accesses with other parts of the code and whether the combined TB will use more shared memory than an original TB. To illustrate the proposed approaches, we use the FFT as a running example. Figure 5.2 shows the pseudo code of the kernel function, which implements a
1K-point FFT through a sequence of 4-point FFT (FFT4 functions) and data interchange through shared memory (loadFromSM and saveToSM functions). In loadFromSM, threads load data from shared memory. In saveToSM threads save data to the shared memory. ‘__syncthreads()’ is used to ensure the order of the shared memory accesses. We also include the sequence number as a parameter of FFT4, loadFromSM and saveToSM functions to show the different parts of the code. With this implementation, each TB has 64 threads and uses 8736-Byte shared memory (8192 Bytes for data, additional bytes for padding to avoid bank conflicts and a few bytes reserved by CUDA). This shared memory usage is obtained from the NVCC compiler.

```
1. int v_tb_id = threadIdx.x/64; //virtual thread block id
2. loadFromGlobal();
3. FFT4(0);
4. if (v_tb_id==0) saveToSM(0); //def. from threads in v_tb_0
5. __syncthreads();
6. if (v_tb_id==0) loadFromSM(0); //use. from threads in v_tb_0
7. __syncthreads();
8. if (v_tb_id==1) saveToSM(0); //def. from threads in v_tb_1
9. __syncthreads();
10. if (v_tb_id==1) loadFromSM(0); //use. from threads in v_tb_1
11. FFT4(1);
12. ....
13. FFT4(4);
14. writeToGlobal();
```

Figure 5.3. The pseudo code of a 1k-point FFT implementation using VTB. Each thread block uses 8736-Byte shared memory and there are 128 threads in each thread block.

5.4.1 Virtual Thread Block (VTB)

In this approach, we first isolate the part(s) of a kernel function that accesses shared memory variables. Secondly, we combine two original TBs into a new one. Here, we refer to an original TB as a virtual TB. In other words, after TB combination, one TB contains two virtual TBs. Thirdly, we introduce the control flow “if(v_tb_id==0)” and “if(v_tb_id ==1)” to manage which virtual TB will
access the shared memory at a time. The amount of the required shared memory of the combined TB remains the same as either of the virtual TBs.

For the FFT code example, the code after we apply VTB is shown in Figure 5.3. The ‘if-statements’ on lines 4, 6, 8, and 10 are introduced to ensure that only one virtual TB is accessing the allocated shared memory at a time. The ‘syncthreads()’ function on line 7 implicitly marks the last use of the shared memory of virtual TB 0 so that virtual TB 1 can use the shared memory immediately afterwards.

Next, we illustrate the reason why our proposed VTB can improve the GPU throughput and also highlight its overhead. Assuming a GPU with 16kB shared memory in each SM, since each TB requires more than 8kB shared memory, two TB dispatched to the same SM have to execute back to back with the code in Figure 5.2. This execution process is shown in Figure 5.4a. For the purpose of clarity, in Figure 5.4 we only show the execution time corresponding to the global memory access, the first 4-point FFT and the data exchange via the shared memory. The remaining code in the kernel function simply repeats 4-point FFT and data exchange multiple times. With the code in Figure 5.3, the combined TB is equivalent to the two original TBs. Due to the increased TLP, the execution time of the function loadFromGlobal() and FFT4() of 128 threads is significantly less than the back-to-back execution of the same functions for 64 threads, as shown in Figure 5.4b. However, to control the accesses to shared memory between the two virtual TBs, additional synchronization functions are added to ensure correctness. Besides the latency to perform such ‘__syncthreads()’ functions, the barrier also limits the compiler’s capability to schedule instructions across the barriers, which may result in reduced instruction-level parallelism (ILP) and additional register usage. The added control flow instruction “if(v_tb_id==0)” has minimal overhead as it does not generate any control divergence within a warp since all 64 threads in the same virtual TB will follow the same direction and each warp has 32 threads on NVIDIA GPUs. The global memory access functions ‘loadFromGlobal’ and ‘writeToLocalGlobal’ benefit from VTB as the increased TLP translate to increased memory-level parallelism (MLP).

From Figure 5.4, we can also see that when a virtual TB accesses the shared memory, the other virtual TB is forced to be idle due to the control flow and the ‘__syncthreads()’ functions. Our proposed second and third software approaches address this limitation and we include the execution time information of these approaches in Figure 5.4c and 5.4d for comparison. We discuss these two approaches in detail in Sections 5.2 and 5.3.
Figure 5.4. A comparison of execution time of the baseline to our proposed software approaches: (a) the baseline, (b) VTB, (c) VTB_pipe, and (d) CO-VTB.

Note that although Figure 5.3 and Figure 5.4 show the case of combining two original TBs into one, we can apply the same principle to combine more than two TBs. The optimal number of TBs to combine is dependent on how many concurrent threads can run on an SM. Typically, combining two TBs is sufficient to reap most performance benefits. Among all the benchmarks in our study, only
MarchingCubes (MC) benefits from combining more than two TBs (we combined 4 TBs for MC using our proposed VTB approach).

5.4.2 Pipelined Virtual Thread Block (VTB_PIPE)

As discussed in Section 5.1, VTB combines two virtual TBs into a larger one and it ensures that only one virtual TB is accessing the shared memory by forcing the other virtual TB to be idle. To reduce such idle cycles, we propose to overlap computation with shared memory accesses. To do so, we make the first virtual TB to run faster than the second one using an ‘if(v_tb_id==0)’ statement. Then, when the first virtual block reaches the code section of shared memory access, the second virtual TB continues its computation instead of being forced idle. When the second virtual block reaches the code section of shared memory accesses, the first will continue to run ahead. This process is similar to letting the two virtual TBs to go through a pipeline. Therefore, we refer to this approach as pipelined VTB (VTB_pipe).

For the FFT example, the code after we apply our proposed VTB_pipe is shown in Figure 5.5. From Figure 5.5, we can see that initially the two virtual TBs will both execute the ‘loadFromGlobal()’ function. Then, the ‘if(v_td_id==0)’ statements on lines 3 and 4 as well as the ‘__syncthreads()’ on line 5 enable virtual TB 0 to execute the ‘FFT4()’ and ‘saveToSM’ functions, making it running ahead of virtual TB1. The code on line 6 and line 7 shows the overlapping between the function ‘loadFromSM()’ of virtual TB0 and the ‘FFT4()’ function of virtual TB1. Since virtual TB1 is lagging behind, when it reads from the shared memory via ‘loadFromSM()’ on line 11, the virtual TB0 proceeds to compute its next 4-point FFT, the ‘FFT4()’ on line 12. The execution process is shown in Figure 5.4b. Due to the overlapping between shared memory accesses and computation, we can reduce the idle cycles experienced by virtual TBs.
Figure 5.5. The pseudo code of a 1k-point FFT implementation using VTB_pipe. Each thread block uses 8736-Byte shared memory and there are 128 threads in each thread block.

The complexity of VTB_pipe, however, is that we may need to partition the non-shared memory access code section to create small computational/global memory access tasks so that they can overlap with shared memory accesses. The ideal case is that the small computational tasks have similar execution latency to the shared memory accesses and can completely utilize the otherwise idle cycles. In the FFT example, the FFT4 is a convenient choice and does not require such partition. For other benchmarks such as Histogram (HG), loop peeling is used to create such a computational/global memory access task to overlap with the shared memory accesses. Similar to the VTB approach, we can choose to combine more than two original TBs. However, synchronization among more than two TBs becomes difficult to manage. Therefore, we choose not to combine more than two TBs for our VTB_pipe approach.

5.4.3 Collaborative Virtual Thread Block (CO_VTB)

In both VTB and VTB_pipe, a few original TBs are combined to time multiplex the allocated shared memory, thereby significantly improving TLP. For some applications, the TLP improvement is
sufficient to hide instruction execution latencies. For others, there exist additional opportunities. As discussed in Section 5.3, if the shared memory size is not a multiple of the shared memory usage of a TB, part of shared memory is always wasted. Neither VTB nor VTB_pipe addresses this issue as they do not alter the shared memory usage of a TB. To effectively utilize such otherwise wasted shared memory, we propose to let two TBs to collaboratively utilize the shared memory and refer to this approach as collaborative virtual thread blocks (CO-VTB). In CO-VTB, we partition the shared memory usage of a TB into two parts, private and public, and apply the VTB (or VTB_pipe) approach only on the public part. For example, if the original shared memory usage of a TB is 9kB, an SM with 16kB shared memory can only host 1 TB. If we partition the shared memory usage of a TB into a 7kB private part and a 2kB public part, when we combine two TBs, each uses 7kB private shared memory each (a total of 14kB) and both time multiplex the 2kB public shared memory, thereby utilizing the 16kB shared memory effectively. Figure 5.4d illustrates the execution of our proposed CO-VTB approach.

Next, we use the benchmark, MatchingCube, to show the code changes for CO-VTB. For the benchmark FFT, CO-VTB involves too much code change, which incurs high performance overhead. The simplified pseudo code of the baseline MatchingCube kernel is shown in Figure 5.6a. From the code, we can see that each TB has 32 threads and uses 9216-Byte (=24*32*3*4) shared memory. The code after we apply CO-VTB is shown in Figure 5.6b. Now, one TB has 2 virtual TBs and there are 64 threads in a TB. The shared memory array is partitioned into two parts: the private arrays ‘vertlist_v0’ and ‘vertlist_v1’, which are combined into a single array ‘vertlist’, and the public array ‘vertlist2’, which is multiplexed by the two virtual TBs. Either private array has the size of 6144(=16*32*3*4) Bytes and the public array size is 3072(=8*32*3*4) Bytes. So, the overall shared memory usage of a TB becomes 15360 (= 2*6144+3072) Bytes. The register variable ‘reg’ is introduced to temporarily hold the data to be written to the public part of the shared memory. Additional code is inserted to check the array index (the variable ‘edge’) to determine whether the data resides in the private or public part of shared memory and then either the array ‘vertlist’ (private) or the array ‘vertlist2’ (public) is used accordingly.

As shown from the code example in Figure 5.6, there is overhead involved in the CO-VTB approach, including additional register variables and additional code. For kernel functions like FFT, the complex array access patterns introduce too much overhead when a shared memory array is partitioned to private and public parts. Therefore, CO-VTB is utilized selectively, only for arrays with
relatively simple access patterns. Due to this complexity, we also choose not to combine more than two TBs.

Figure 5.6. The simplified pseudo code of MarchingCubes, (a) the baseline kernel code; (b) the code after we apply CO-VTB.
5.5 Shared memory Multiplexing: A Hardware Solution

As discussed in Section 5.4, our proposed software approaches improve TLP by merging original TBs and explicitly managing the shared memory accesses among them. The advantage of the software approaches is that they work well with current GPUs. The disadvantage, however, is the overhead introduced to manage the shared memory. In this section, we propose a hardware solution to managing shared memory.

In GPUs, the TB dispatcher dispatches TBs onto SMs. For each SM, it maintains a shared memory management (SMM) table, as shown in Figure 5.7. The SMM table has multiple entries and each entry keeps three fields, the TB id, the size, and the starting address. When a TB is to be dispatched to an SM, the TB dispatcher goes through the SMM table of the SM to determine whether there is enough free shared memory. If so, the dispatcher allocates the required shared memory by filling an entry in the SMM table with the TB id and setting its size field to the required shared memory size of the TB. The starting address field is determined and then passed to the TB so that every shared memory access in the TB will use this starting address as the base address. When a TB finishes execution, the dispatcher releases the allocated shared memory by invalidate the corresponding SMM table entry. Since the shared memory is allocated through the lifetime of a TB, we refer to such shared memory management as ‘static’ allocation and de-allocation.

To enable dynamic shared memory management, we propose to extend the TB dispatcher so that the shared memory management is exposed to and can be controlled by software. Since shared memory allocation and de-allocation affect all the threads in a TB, we propose to associate shared memory management with the existing ‘__syncthreads()’ function and the new syntax of the function becomes ‘__syncthreads(int opt, unsigned &base_addr, unsigned size)’. It still serves as a barrier to synchronize all the threads in a TB. When the parameter ‘opt’ is 1, it invokes the TB dispatcher to allocate the shared memory for ‘size’ bytes. The TB dispatcher uses the same allocation process by going through the corresponding SMM table. If there is enough free shared memory, an entry in the SMM table is updated and its ‘starting address’ field is passed to the ‘base_addr’ variable to be used by subsequent shared memory accesses. If there is no sufficient shared memory to be allocated, the TB will be stalled until another TB frees its allocated shared memory. If the parameter ‘opt’ is ‘-1’, the TB dispatcher performs shared memory de-allocation using the ‘base_addr’ parameter. It searches the SMM table entries to find the matching ‘starting address’ with the ‘base_addr’ and invalidates the
Table entry. If the parameter ‘opt’ is ’0’, the other two parameters (‘base_addr’ and ‘size’) are ignored and ‘__syncthreads’ operates as a regular barrier. To simplify the design, we choose not to allow nested shared memory allocation so as to avoid any potential deadlock issue. We also require that for a kernel function, the size of dynamic allocation and de-allocation to be the same so as to avoid fragmentation.

<table>
<thead>
<tr>
<th>Valid</th>
<th>TB id</th>
<th>Size</th>
<th>Starting Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>4kB</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>4kB</td>
<td>4096</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

**Figure 5.7. A shared memory management (SMM) table.**

The code change to utilize our proposed dynamic shared memory management is very little. It only needs a shared memory allocation at the beginning and de-allocation at the end of each memory access code region. For the FFT kernel, the code after such changes is shown in Figure 5.8.

```c
loadFromGlobal();
FFT4(0);
__syncthreads(1, &base_addr, 8376); //allocation
saveToSM(0); //define by multiple threads in a TB
__syncthreads(0, 0, 0); //synchronization
loadFromSM(0); //use by multiple threads in a TB
__syncthreads(-1, &base_addr, 8376); //de-allocation
FFT4(1);
....
writeToGlobal();
```

**Figure 5.8. The pseudo code for 1k-point FFT kernel using the hardware supported dynamic shared memory management. Each TB allocates and de-allocates 8376-byte shared memory and contains 64 threads.**
The dynamic shared memory allocation and de-allocation in the FFT kernel shown in Figure 5.8 enables an SM to exploit higher degrees of TLP. Figure 5.9 illustrates this effect with two TBs running on an SM. Although the shared memory (16kB) on the SM is not large enough for the aggregate requirement from the two TBs (2x8376=16752B), our proposed dynamic allocation enables them to run concurrently and ensures that the two allocation calls will be served one after the other.

From Figure 5.9, it can be seen that the key performance advantages of our dynamic shared memory management include: (1) higher degrees of TLP to hide instruction latencies, and (2) reduced overhead of '__syncthreads()' as the barrier is limited to a TB and doesn’t affect other TBs. In comparison, in our software approaches, such a barrier will affect both virtual TBs. Furthermore, dynamic allocation and de-allocation naturally enables overlap between shared memory accesses of one TB and non-shared memory code in another as long as they do not reach allocation at the same time. This is the reason why we do not need the code changes of our VTB or VTB_pipe approaches.

With the hardware supported shared memory management, we do not need to combine TBs. A TB can be dispatched to an SM as long as other resource requirements such as registers are satisfied. As a result, there might be too many TBs dispatched to an SM. We propose to use a counter to track how many TBs are running on an SM and stall TB dispatching when this counter reaches a threshold. If we denote the maximum number of TBs that can be dispatched to an SM using the static shared memory management as $K$, the threshold setting of $K+2$ or $K+3$ achieves good performance (See

**Figure 5.9. Two TBs running concurrently on an SM using dynamic shared memory management.**
In other words, allowing an SM to run 2 or 3 more TBs concurrently usually improves TLP sufficiently.

With our proposed hardware solution, the TB dispatcher can support both static and dynamic shared memory management. We propose to let either the run-time or compiler to determine which mechanism to be used. If the static management is selected for the purpose of quality of service, the dynamic management instructions are ignored. Static and dynamic shared memory management can also be used together to support collaborative TBs, similar to the idea exploited in CO-VTB. We refer to this hardware supported collaborative TB approach as CO-HW. Like CO-VTB, we need to change the kernel code to partition the shared memory usage into the private part and the public part. Unlike CO-VTB, we do not need to use VTB or CO-VTB on the public part. Instead, we insert dynamic allocation and de-allocation instruction to multiplex the public part. When a kernel is launched to a GPU, the compiler or the run-time provides the sizes of both the private part and the public part to the TB dispatcher. Static shared memory management is used for the private part and dynamic shared memory management is used to multiplex the public part. For example, a TB originally uses 6kB shared memory and an SM with 16kB shared memory can host two such TBs using static shared memory management. After partition, a TB uses 4kB private and 2kB public shared memory. With 16kB shared memory, three TBs can run concurrently using a total of 12kB (=4kBx3) private shared memory. The remaining 4kB is used as public shared memory among the three TBs. The same SMM table is used to manage shared memory as shown in Figure 5.10, where the first three entries are allocated when the three TBs are dispatched and the last two are allocated/de-allocated with the ‘__syncthreads(opt, &base_addr, size)’ instructions.

<table>
<thead>
<tr>
<th>Valid</th>
<th>TB id</th>
<th>Size</th>
<th>Starting Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>4kB</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4kB</td>
<td>4096</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4kB</td>
<td>8192</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2kB</td>
<td>12288</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2kB</td>
<td>14436</td>
</tr>
</tbody>
</table>

Figure 5.10. The SMM table manages shared memory usage for collaborative TBs. The first three entries are for the private shared memory of the 3 TBs. The last two entries are for the public shared memory.
Due to static management, the private part is allocated when a TB is dispatched and is de-allocated when it completes execution. The public part is managed based on the dynamic allocation and de-allocation instructions. The only constraint, which the TB dispatcher enforces, is that when a TB is dispatched, the total amount of the private parts of currently running TBs in an SM cannot exceed (the size of overall shared memory – the size of the public part of a TB). The purpose is to ensure that there is at least one set of public shared memory available to be used among the TBs.

Our proposed hardware solution simply exposes the existing shared memory management in the TB dispatcher to the software and enables it to be controlled by the extended ‘__syncthreads()’ instructions. The code change is to insert ‘__syncthreads(1, &base_addr, size)’ at the beginning of shared memory access regions and ‘__syncthreads(-1, &base_addr, size)’ at the end. For CO-HW, this region is where the public shared memory part is accessed. Therefore, we argue that this solution has low overhead in both hardware and software changes. The effectiveness is evaluated in Section 5.7.2.

5.6 Experimental Methodology

To evaluate our proposed software approaches, we use both NVIDIA GTX 480 and NVIDIA GTX285 GPUs with CUDA SDK 4.0. Because the shared memory size is configurable on GTX 480 GPUs, we present two sets of results: one with 48kB shared memory and the other with 16kB shared memory. As discussed in Section 5.3, the focus of our proposed approaches is the category of applications, which have low TLP due to the limited capacity of shared memory in an SM. Therefore, among all the workloads in Table 1, our experiments do not include MM, TP, and RD as they have high TLP already and are not affected by our approaches. We implemented our proposed VTB and VTB_pipe using a source-to-source compiler, Cetus [44]. For VTB, the compiler searches defines and uses of shared memory variables among all the threads in a TB. The code between the first define and the last use is treated as a shared memory access region. Re-defines to shared memory variables are used to help to determine the last uses in a region. Then, the compiler generates the code for virtual TB ids and the control flow to determine which virtual TB accesses the shared memory. For VTB_pipe, we add annotations manually to denote the section of code to overlap with shared memory access regions and the compiler generates the final code. For CO-VTB, we generate the code manually due to the complexity of partitioning shared memory variables into private and public parts.
Also, as discussed in Section 5.5.3, not all workloads are suitable to CO-VTB due to the associated overheads. Among the workloads, we applied CO-VTB to MC, STO, and HG.

To model our proposed hardware support, we extend the GPGPU sim V3.0 simulator [9] to support our proposed dynamic shared memory management instructions (i.e., the extended '__syncthreads()' instruction). The simulator models an NVIDIA GTX285 GPU, which has 16kB shared memory and a 64kB register file on each SM. The off-chip memory frequency is set to 1100MHz. We manually inserted the dynamic allocation and de-allocation instructions in the workloads. For the CO-HW approach, we used very similar code to CO-VTB except the part using virtual TB ids as we do not need to combine two TBs. We also inserted the dynamic allocation and de-allocation instructions surrounding the code region that accesses the public shared memory.

![Figure 5.11. Speedups of the proposed software approaches over the baseline for different GPUs.](image)
5.7 Experimental Results

5.7.1 Evaluation of Software-Based Shared Memory Multiplexing

In our first experiment, we evaluate the performance improvements from our proposed software approaches, VTB, VTB_pipe and CO-VTB. In Figure 5.11, we report the speedups of these three approaches over the baseline implementation. For each benchmark, we examine three GPU configurations, GTX 480 with 16kB shared memory (labeled “GTX480_SM_16K”), GTX 480 with 48kB shared memory (labeled “GTX480_SM_48K”) and GTX 285 with 16kB shared memory (labeled “GTX285_SM_16K”). From the figure, we can make some interesting observations. First, among the three software approaches, CO-VTB achieves highest performance on average, using geometric mean (GM). However, as discussed in Section 5.6, not all workloads are suitable for this approach. VTB_pipe achieves higher performance than VTB on average as it can overlap shared memory accesses with other parts of code. The benchmark, MC, is an exception since the shared memory accesses dominate its execution and the overlapping effect from VTB_pipe fails to offset the overhead. Since different workloads may favor different shared memory multiplexing approaches, in order to achieve the best performance, we can generate three versions of optimized code using VTB, VTB_pipe and CO-VTB and select the best performing one. Secondly, among three GPU configurations, highest performance gains are achieved on GTX 480 with 16kB shared memory and the average speedups are 1.42X, 1.60X, 1.61X for VTB, VTB_pipe and CO-VTB, respectively. If we select the best version for each benchmark, the average performance gain reaches 1.70X. The reason is that compared to GTX285, GTX480 has a higher number of SPs. One instruction from a warp will keep the SP busy for 4 cycles on GTX285 compared 2 cycles on GTX480. Therefore, compared to GTX285, TLP is more critical for GTX480. Thirdly, the increased shared memory capacity on GTX480 with 48kB shared memory enables an SM to host more TBs. The improved TLP in turn reduces the benefit of our proposed software schemes. For example, for the benchmark CV, each TB has 128 threads and uses 8300 Bytes of shared memory. As a result, 16kB shared memory can only host 1 TB while 48kB shared memory can host 5TBs (or 640 threads). Nevertheless, our proposed approaches remain effective and the achieved speedups are 1.09X, 1.22X, 1.14X on average for VTB, VTB_pipe, and CO-VTB, respectively. Selecting the best version for each benchmark provides a 1.26X speedup on average.
As two of our benchmarks, FFT and MV, are implemented in NVIDIA libraries, we compare our implementation of FFT to CUFFT4.0 and MV to CUBLAS4.0 on GTX480. For MV, we keep the width of the input matrix as 1024 and vary the height from 8K to 128K. The reason is that the height of the input matrix determines the number of threads for MV. The throughput comparison to CUBLAS4.0 is shown in Figure 5.12.

In Figure 5.12, the results with label ‘16K_BL’ and ‘48K_BL’ are our baseline implementation running on GTX480 with 16kB shared memory and 48kB shared memory, respectively. Our VTB_pipe results are labeled as ‘16K_VTB_pipe’ and ‘48K_VTB_pipe’ for the two shared memory configurations of GTX480. From the results, we can see that our baseline implementation running on GTX 480 with 48kB shared memory has similar performance to CUBLAS. With our VTB_pipe approach on GTX480 with 16kB shared memory, we can achieve similar performance to CUBLAS. On the GTX480 configuration with 48kB shared memory, our VTB_pipe approach outperforms CUBLAS by up to 74% and 52% on average.

For 1K-point FFT, we use batch execution [25] to evaluate the throughput and vary the batch size from 128 to 2048. The throughput results are reported in Figure 5.13.
Figure 5.13. Performance comparison of FFT among the baseline (xK_BL), VTB_pipe (xK_VTB_pipe) and CUFFT 4.0 on GTX 480. ‘xK’ denotes the size of shared memory.

From the figure, we can see that our baseline implementation running on GTX480 with 16kB shared memory outperforms CUFFT [54] for small batch sizes and not as good as CUFFT for large batch sizes. With the 48kB shared configuration, our baseline implementation consistently outperforms CUFFT. The average throughput of ‘48K_BL’ is 168.3 GFLOPS compared to the average of 72.4 GFLOPS throughput of CUFFT. Our VTB_pipe further improves the throughput by up to 33% and achieves the average throughput of 205.9 GFLOPS (a 2.84X speedup over CUFFT).

5.7.2 Evaluation of Hardware-Supported Shared Memory Multiplexing

To evaluate the effectiveness of our proposed hardware solution, we first measure the performance of the baseline implementation (i.e., the static shared memory management) and the one with hardware support for dynamic shared memory management. Here, we use execution time (in the unit of cycles) rather than instruction per cycle (IPC) since we insert the dynamic allocation and de-allocation instructions into the code. The speedups of our hardware-supported dynamic shared memory management (labeled ‘HW’) over the baseline are shown in Figure 5.14. We also report the performance results of CO-HW, in which the shared memory usage is partitioned into statically managed private and dynamically managed public parts, in the figure (labeled ‘CO_HW’).

From Figure 5.14, we can see that our hardware supported dynamic allocation and de-allocation can significantly improve the performance, up to 2.34X and 1.53X on average, over the baseline static shared memory allocation. Similar to the software-based CO-VTB approach, we manually modified the code of MC, STO and HG for CO-HW. From Figure 5.14, it can be seen that CO-HW achieves up
to 1.88X and an average of 1.42X speedups over the baseline. As discussed in Section 5.5, our hardware support dynamic shared memory management (‘HW’) eliminates some overheads of VTB or VTB\_pipe. As a result, between HW and CO-HW, CO-HW remains more effective for MC but not for STO and HG. In contrast, for their software counterparts, CO-VTB typically performs much better than VTB or VTB\_pipe.

![Figure 5.14. Speedups of hardware-supported dynamic shared memory management over baseline.](image)

As discussed in Section 5.5, using dynamic shared memory management, many TBs can be dispatched to an SM if the shared memory is the only resource bottleneck. This may generate too much contention for multiplexing shared memory. A counter scheme is proposed in Section 5.5 to control the number of concurrent TBs that can be dispatched to an SM. If we use $K$ to denote the maximum number of TBs that can be supported using the static shared memory management, we vary the upper bound of this counter from $K+1$ to $K+3$ and show the performance impact in Figure 5.15. In other words, we use dynamic shared memory management to allow an SM to host 1~3 more TBs. From the figure, we can see that on average, hosting 1 more TB (‘$K+1$’) does not provide sufficient TLP. Although hosting 2 or 3 more TBs in an SM shows similar performance, individual benchmarks show different trends. FFT and MC favor more TLP while HG and MV do not. Overall, our results suggest that either $K+2$ or $K+3$ is a fine choice as the maximum number of TBs to be allowed to run concurrently on an SM.
Figure 5.15. The impact of the maximum number of TBs to be allowed to run in an SM.

5.8 Related Work

On-chip shared memory is a critical resource for GPGPU applications. Previous works mainly focus on utilizing shared memory to achieve coalesced memory accesses [35][65][66][67][77] [82][86] [89][90], to provide data exchange among threads [25], to use shared memory as software managed cache [86], etc. Although it is well known that heavy usage of shared memory may limit TLP [71][80] [86], it is common that the benefits of using shared memory overweigh the shortcomings of reduced TLP. As a result, many GPGPU workloads as shown in Section 5.3 have exhibited high shared memory usage. This is also a reason why the latest NVIDIA Fermi architecture (e.g., GTX 480 GPUs) provides larger shared memory and an L1 cache. However, as shown in Section 5.7, the high number of SPs in an SM in GTX480 (and even higher number of SPs in GTX680) makes TLP more important to hide instruction execution latencies. In contrast, our work improves TLP without sacrificing the usage of shared memory.

5.9 Conclusion

In this chapter, we propose novel software and hardware approaches to multiplex shared memory. Our approaches are based on our observation that for the GPGPU applications with heavy use of shared memory, the duration of time, when the shared memory is utilized, is actually low. Our experimental results confirm that the shared memory is utilized for only 25.6% of the execution time of a TB. Therefore, there exist significant opportunities to time multiplex shared memory. Among our
software approaches, VTB is simplest and it combines two TBs into a new one and adds control flow to ensure only one original TB accesses the shared memory at a time. VTB_pipe reduces the performance overhead of VTB by overlapping non-shared memory access regions (e.g., computation or global memory accesses) with shared memory accesses. CO-VTB partitions the shared memory data into a private part and a public part and only applies VTB/VTB_pipe upon the public part. Our proposed hardware support essentially exposes the existing shared memory management to software and enables software to control when to perform allocation and de-allocation. Our experimental results show that our proposed software schemes improve the performance significantly on current GPUs. We evaluate our hardware solution using the GPGPUsim simulator and the results show that it improves the performance remarkably with very little change in GPGPU code.
Chapter 6

CPU-Assisted GPGPU on Fused CPU-GPU Architecture

6.1 Introduction

The integration trend of CMOS devices has led to fused architectures, in which the central processing units (CPUs) and graphics processing units (GPUs) are integrated onto the same chip. Recent examples include Intel's sandy bridge [69], on which CPUs and GPUs are on one chip with a shared on-chip L3 cache, and AMD accelerated processing unit (APU) [5] on which both on-chip CPUs and GPUs share the same off-chip memory [16]. Such heterogeneous architectures provide the opportunity to leverage both the high computational power from GPUs for regular applications and flexible execution from CPUs for irregular workloads. In this chapter, we assume that the fused CPU-GPU architecture has a shared L3 cache and shared off-chip memory between CPUs and GPUs and we propose a novel approach to collaboratively utilize the CPU and GPU resources. In our proposed approach, called CPU-assisted GPGPU (general purpose computation on graphics processor units), after the CPU launches a GPU program, it starts a pre-execution program to prefetch the off-chip memory data into the shared L3 cache for GPU threads.

Our proposed CPU-assisted GPGPU works as follows. First, we develop a compiler algorithm to generate the CPU pre-execution program from GPU kernels. It extracts memory access instructions and the associated address computations from GPU kernels and then adds loops to prefetch data for concurrent threads with different thread identifiers (ids). The update of the loop iterators provides a flexible way to select/skip thread ids for prefetching (see Section 6.4). Secondly, when the GPU program is launched, the CPU runs the pre-execution program. Such pre-execution is expected to warm up the shared L3 cache for GPU threads since (1) the pre-execution program only contains the memory operations (and address calculations) but not floating point/ALU computations; and (2) the CPU runs at a higher frequency and is more aggressive in exploiting instruction-level parallelism.
(ILP). To make the proposed pre-execution effective, it is critical to control the timing of the prefetches since they need to be issued early enough to hide memory access latencies while at the same time not so early that the prefetched data might be replaced before being utilized. We propose two mechanisms to achieve this through the loop iterator update code in the pre-execution program. The first is an adaptive approach, which requires a new CPU instruction to inquire the performance counter of L3 cache hits periodically so as to adjust the loop iterator update. The insight is that if there are too many L3 hits experienced by the CPU pre-execution program, it means that CPU is not effective in fetching new data into the L3 cache. As a result, the CPU needs to run further ahead by increasing the amount of loop iterator update. On the other hand, if there are too few L3 cache hits, meaning that the CPU pre-execution program may skip too many threads, it reduces the loop iterator update so as to select more thread ids for prefetching. The second is a static approach, which determines the best loop iterator update value based on profiling and does not require any new hardware support.

In summary, this chapter makes the following contributions: (1) we propose to utilize the otherwise idle CPU to assist GPGPU through pre-execution; (2) we propose compile algorithms to generate the CPU pre-execution program from different types of GPU kernels; (3) we propose simple yet effective approaches to control how far the CPU code runs ahead of GPU threads; and (4) we implemented our proposed schemes by integrating and modifying the MARSS X86 [48] and the GPGPUsim [9] timing simulators and our results show that our proposed CPU pre-execution can improve the performance of GPU programs by up to 113% (126%) and 21.4% (23.1%) on average using adaptive iterator update (fixed iterator update). The cost of achieving such performance gains is nominal: the average instruction overhead of the CPU pre-execution program is 0.74% (0.69%) of the number of instructions executed by GPU using adaptive iterator update (fixed iterator update).

The remainder of the chapter is organized as follows. In Section 6.2, we present a brief background on fused CPU-GPU architectures. In Section 6.3, we present our modeling of fused CPU-GPU architecture and our experimental methodology. Section 6.4 discusses our proposed CPU-assisted GPGPU in detail. The experimental results are presented in the Section 6.5. Related work is discussed in Section 6.6. Finally, Section 6.7 concludes the chapter.
6.2 Background

6.2.1 Fused CPU-GPU architectures

Advances in the CMOS technology make it possible to integrate multi-core CPUs and many-core GPUs on the same chip, as exemplified with the latest AMD’s accelerated processing units (APUs) and Intel’s Sandy Bridge processors. Figure 6.1 shows such a fused architecture with a shared L3 cache and shared off-chip memory. We ignore the introduction of the microarchitecture of the SM in Figure 6.1 here, since it is the same as the microarchitecture of a SM of a discrete GPU presented in Chapter 2.1 but is running at lower frequency.

![Figure 6.1: A fused CPU-GPU architecture with a shared on-chip L3 cache and off-chip memory.](image)

Compared to high-end discrete GPUs, the GPUs on current fused architectures have less computation throughput and lower memory access bandwidths. However, with the shared off-chip memory, fused architectures eliminate the costly CPU-GPU data transfers [16] and the existence of the shared on-chip L3 cache opens more opportunities for close collaboration between GPUs and CPUs. Compared to CPUs, the on-chip GPUs still deliver high computational power. For example, on AMD E2-3200 APU, the GPU module (HD6370D) has the throughput of 141.8 GFLOPS while the CPU has a throughput of 38.4 GFLOPS with SSE [75]. In this chapter, we propose a new approach to
collaboratively utilize both CPU and GPU resources on fused architectures efficiently to achieve high performance GPGPU.

6.3 Architectural Modeling and Experimental Methodology

In this chapter, we model the fused architecture as shown in Figure 6.1. To build our simulator infrastructure, we use the full system X86 timing simulator MARSSx86 [48] for the CPU part and the GPGPUSIM [9] for the GPU part. To merge two simulators to model the fused architecture, we consider the CPU simulator as the host, meaning that the GPU simulator is invoked from the CPU simulator. In every few CPU cycles determined by the frequency ratio of CPU over GPU, the CPU simulator invokes a GPU cycle. We also ported the DRAM model in the GPGPUSIM into marssx86.

To enable the full system simulator to run CPU code and GPU code collaboratively, we partition the memory space of the fused CPU-GPU architecture into two parts: the lower address memory space that is used solely by CPU and the upper address memory space that can be accessed by both GPU and the CPU. For example, if we allocate 256MB memory for the whole system, we reserve the upper 128MB memory by passing “-mem=128M” as a parameter to boot the Linux operating system so that the operating system only uses the lower 128MB memory and reserves the upper 128MB memory for GPU. The GPU accesses the DRAM directly with physical addresses starting from 128MB. If a CPU application needs to access the upper memory space, we need to first use the Linux function ‘ioremap’ to map the upper 128MB memory space into the operating system as a memory device. Then, applications in user space can use the ‘mmap’ system call to map the memory device into the user virtual memory space. This way, applications in user space and GPU programs can access the same DRAM memory. As a side effect of our simulator infrastructure, the CPU memory accesses have higher overheads than GPU memory accesses since applications in CPU user space need to first perform address mapping to get physical addresses. However, as shown in Section 6.5, our proposed CPU pre-execution can achieve significant performance improvement even with our pessimistic handling of CPU memory accesses compared to GPU memory accesses.

The parameters in our simulator are set up to model AMD APU E2-3200 [5], except the shared L3 cache. For the CPU part, we model a 4-way-issue out-of-order CPU running at 2.4 GHz with a 128KB L1 cache and a 512KB L2 cache. For the GPU part, we model a GPU with 4 SMs and each SM contains 32 SPs running at 480M Hz. Each SP can deliver up to 2 flops (a fused multiplication
and add) every cycle. As a result, the overall GPU computation power in our model is about 122.8GFLOPS. The register file and the shared memory in each SM are 32kB (8k registers) and 16KB, respectively. Each SM can support up to 768 active threads. In our experiments, we also vary the key parameters to explore the design space of the on-chip GPU. Our dram memory model has 8 memory modules and every module support a bus width of 32 bits. So, the bandwidth of DRAM memory in the simulator is 600M Hz * 32 bits * 8=19.2GB/s. Because the frequency of GPU is 1/5 of the CPU frequency, the simulator executes one GPU cycle every 5 CPU cycles and a DRAM cycle every 4 CPU cycles. We also add the L3 cache to the simulator, which is 4MB and is shared by the CPU and the GPU. The L3 cache hit latency for CPU is 40 CPU cycles while the L3 hit latency for GPU is 20 GPU cycles (i.e., 80 CPU cycles). The difference is to account for the fact the CPU core is located closer to the L3 cache than GPU and GPU is less latency sensitive than CPU.

### Table 6.1. Benchmarks for CPU-assisted GPGPU

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Input sizes</th>
<th>Number of threads</th>
<th>Threads per thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blackscholes (BSc)</td>
<td>1M</td>
<td>1M</td>
<td>128</td>
</tr>
<tr>
<td>Vector-Add (VD)</td>
<td>1M</td>
<td>1M</td>
<td>128</td>
</tr>
<tr>
<td>Fast Fourier Transform (FFT)</td>
<td>1M</td>
<td>512K</td>
<td>128</td>
</tr>
<tr>
<td>Matrix Multiplication (MM)</td>
<td>512x512</td>
<td>256K</td>
<td>16x16</td>
</tr>
<tr>
<td>Convolution (Con)</td>
<td>1kx1k &amp; 3x3</td>
<td>1M</td>
<td>16x16</td>
</tr>
<tr>
<td>Transpose matrix vector multiplication (TMV)</td>
<td>512x2048</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>BitonicSort (BS)</td>
<td>2M</td>
<td>1M</td>
<td>256</td>
</tr>
<tr>
<td>MonteCarlo (MC)</td>
<td>256K</td>
<td>128K</td>
<td>256</td>
</tr>
<tr>
<td>Matrix-vector multiplication (MV)</td>
<td>16x65k</td>
<td>65K</td>
<td>256</td>
</tr>
</tbody>
</table>

The benchmarks used in our experiments are listed in Table 6.1. Among the benchmarks, Blackscholes, Vecadd, and Montecarlo are from NVIDIA SDK [52]. BitonicSort and Convolution are from AMD SDK [4]. Matrix multiplication is from [9]. We implemented and optimized transpose-
Our proposed compiler algorithms to generate CPU code from GPU kernels (See Section 6.4) are implemented using a source-to-source compiler infrastructure, Cetus [44].

6.4 CPU-Assisted GPGPU

In our proposed CPU assisted GPGPU, after the CPU launches the GPU kernel, it starts running a pre-execution program to prefetch data into the shared L3 cache for GPU threads. Although we use the same compiler algorithm to generate CPU pre-execution programs from different GPU kernels, for the purpose of clarity, we classify GPU kernels into two types, (a) lightweight workload in a single GPU thread (LWST), and (b) heavyweight workload in a single GPU thread (HWST), and discuss the generation of CPU pre-execution programs separately. The difference between the two types is that HWST GPU kernels have one or more loops, which contain global memory accesses while LWST kernels do not.

In this section, we first present our compiler algorithm to generate the CPU pre-execution program from LWST GPU kernels. Next, we discuss the mechanisms to control how far the CPU pre-execution can run ahead of GPU threads. Then, we present the generic compiler algorithm to generate the CPU pre-execution program from LWST/HWST GPU kernels.

6.4.1 Generating the CPU Pre-Execution Code from LWST GPU Kernels

For LWST GPU kernels, our proposed compiler algorithm to generate the pre-execution program is shown in Figure 6.2.
1. For a GPU kernel, extract its memory operations and the associated address computations and put them in a CPU function; replace thread id computation with an input parameter.

2. Add a nested loop structure into the CPU code to prefetch data for concurrent threads.
   a. The outer loop traverse through all TBs. The iterator starts from 0 and the loop bound is the number of thread blocks of the GPU program. The iterator update is the number of concurrent TBs, meaning the number of TBs that can run concurrently on the GPU.
   b. The second-level loop traverses through concurrent threads. The loop iterator starts from 0 and the loop bound is the number of concurrent threads (which is the product of TB size and the number of concurrent TBs). The iterator update is set as a product of three parameters, unroll-factor, batch_size, and skip_factor.

**Figure 6.2. The compiler algorithm to generate CPU pre-execution program from LWST GPU kernels.**

As shown in Figure 6.2, the algorithm contains two parts. First, it extracts all the memory access operations and the associated address computations from a GPU kernel and puts them in a CPU function. The store operations are converted to load operations. The thread id is converted to a function parameter. Secondly, the compiler adds loops so as to prefetch data for concurrent threads. Since GPU threads are organized in thread blocks (TBs), a nested loop is inserted. The outer loop is used to traverse through all TBs. The iterator starts from 0 and is updated with the number of concurrent TBs, which is determined by the resource usage (registers, shared memory, and the TB size) of each TB and the GPU hardware configuration. The resource usage information is available from the GPU compiler such as nvcc and the hardware information is dependent on the target platform. The second-level loop is used to traverse through all concurrent threads in these concurrent TBs and the iterator is used to compute the thread ids, for which the data will be prefetched. As shown in Figure 6.2, the iterator update is set as a product of three parameters (unroll_factor * batch_size * skip_factor). The last two are used in our proposed mechanisms to control how far the pre-execution program should run ahead of GPU threads (See Section 6.4.2). The unroll factor is used to boost the memory requests from the CPU. Before dissecting this parameter, we first illustrate our compiler algorithm using a vector-add GPU kernel, which is an LWST kernel as it does not have any loops containing global memory accesses. Both the GPU code and the CPU pre-execution code are shown in Figure 6.3.
As shown in Figure 6.3, the function ‘memory_fetch_for_thread’ is a result of extracting the memory accesses from the GPU kernel and can be used to prefetch data for the thread with thread id ‘n’. The memory update operation on ‘C[n]’ is converted to a load operation. The loaded values are summed together so as to avoid the compiler eliminating this function as dead code when the CPU program is compiled. The outer loop in the function ‘cpu_prefetch’ is the one that traverse all TBs. The loop bound is ‘N_tb’ is computed as (N / TB size), i.e., number of TBs. The iterator update ‘concurrent_tb’
is the number of TBs that can run concurrently on the GPU. The second loop with the iterator ‘i’ is introduced to prefetch data for concurrent threads. The loop iterator ‘i’ is used to compute the thread ids, for which the data will be prefetched. Since only those thread ids: (thread_id + skip_factor * 0), (thread_id + skip_factor * 1), …, (thread_id + skip_factor * (unroll_factor – 1)) will be used for prefetching, the variable ‘skip_factor’ determines the number of threads to be skipped before a thread id is used for prefetching. We initialize this variable ‘skip_factor’ to be ‘L3 cache line size / the size of float’ (16 according to our model) so that we do not waste CPU instructions prefetching the data from the same cache line. Similarly, if the next-line prefetching is enabled, ‘skip_factor’ is initialized to ‘2 x L3 cache line size / the size of float’.

In order to make the CPU pre-execution program effective, we need to consider the memory traffic carefully as the GPU in fused GPU-CPU architectures can easily dominate the memory traffic, in which case the CPU prefetching impact will be very limited. The reasons are (1) a GPU has many SPs (128 in our model) and every SP can issue one memory fetch in one GPU cycle. If these fetches miss the GPU caches and cannot be merged, GPU will have a high rate of off-chip memory accesses; (2) GPU is designed to support high degrees of thread-level parallelism (TLP) and independent threads can issue memory requests as long as there are no structural hazards on resources to support outstanding requests such as miss status handling registers (MSHRs) or memory request queues. As a result, although the GPU frequency is slower than the CPU frequency, the number of memory fetches from the GPU can be much larger than those from the CPU. We analyze this effect using the example code in Figure 6.3 and the results are shown in Figure 6.4.

In Figure 6.4, we compare the memory requests generated from the CPU running the pre-execution code with different unroll factors (labeled ‘un_N’). We also enable the L2 cache next-line prefetcher from the CPU side to boost the number of memory requests from CPU (labeled ‘un_N_pref’). The rate of memory requests generated by the GPU is also included for comparison. From the figure, we can see that (1) increasing the unroll factor increases the number of memory requests significantly. With a unroll factor of 16, the memory requests are about 1.5X of those with a unroll factor of 1. (2) Prefetching is also important to maximize the memory requests from the CPU side. An unroll factor of 8 combined with next line prefetching achieve good utilization of the request queue at the CPU L2 cache, which is the reason why we set the value of unroll factor as 8 in Figure 6.3. (3) Given the high number of GPU SPs, GPU memory requests are about 3X compared to the CPU memory requests using our default ‘un_8_pref’ configuration. We also tried with increasing the number of cache lines
which are prefetched in L2 to further increase the number of memory requests from CPU, our results with prefetching 2 or 4 cache lines only show less than 0.2% speedup compared to the next-line prefetcher. Therefore, we use the next-line prefetcher in our experiments.

![Image of Figure 6.4](image_url)

**Figure 6.4.** Comparing memory requests generated from different versions of CPU code and from the GPU for every 100,000 cycles. “un\_N” means the CPU pre-execution program with the unroll factor of N; “un\_N\_pref” means the CPU pre-execution program with the unroll factor of N and with the CPU L2 cache next-line prefetch enabled.

### 6.4.2 Mechanisms to Control How Far the CPU Code Can Run Ahead of GPU Threads

As shown in Figure 6.2, the CPU pre-execution code primarily contains the memory operations from the GPU kernel. Considering the fact that CPU runs at a higher frequency and employs out-of-order execution to exploit instruction-level parallelism (ILP), we expect that the CPU pre-execution code runs ahead of the GPU kernels, meaning that when the CPU code selects a thread id for prefetching, the corresponding GPU thread has not yet reached to the corresponding memory access instruction. As discussed in Section 6.4.1, the parameter ‘skip\_factor’ determines how many thread ids to be skipped before one is used for prefetching. Adjusting this parameter provides a flexible way to
control the timeliness of the CPU prefetches. Here, we propose two mechanisms to adjust this parameter. The first one is an adaptive approach and the second is a fixed value based on profiling.

In our adaptive approach, we design the update algorithm for the variable ‘skip_factor’ based on the following observations. If the CPU pre-execution program has experienced too many L3 cache hits, it means that the memory requests generated from the CPU are not useful because no new data are brought into the L3 cache and it can be that the GPU threads are running ahead and already brought in the data. Therefore, we need to increase the ‘skip_factor’ to make the CPU run further ahead. On the other hand, if there are too few L3 cache hits for CPU pre-execution, it means that CPU side is running too far ahead and we can reduce ‘skip_factor’ to skip fewer thread ids to generate more prefetches. To determine whether there are too many or too few L3 cache hits, we periodically sample the number of L3 cache hits for the CPU and compare the current sample with the last one. If the difference is larger than a threshold, which is set as a fixed value of 10 (our algorithm is not sensitive to this threshold setting as shown in Section 6.5.4), we need to update the skip_factor. The implementation of this adaptive approach for the code in Figure 6.3 is shown in Figure 6.5. Such update code is inside the second-level loop with iterator ‘i’ in the code shown in Figure 6.3 and is executed after we process a batch of thread ids. In other words, the variable ‘batch_size’ determines how often we update the ‘skip_factor’. In our implementation, batch size is set to 16, meaning that we update the ‘skip_factor’ once we process (16 x 8 x skip_factor) thread ids.

```c
//Accessing an L3 cache Performance counter
ptlcall_cpu_read_value(PTLCALL_CPU_NAME_CPU_HIT_L3, &hitnumber);
if (hitnumber-last_hit>threshold) skip_factor += 32;
else  if (back_dis != skip_factor -32) {
//preventing skip_factor bouncing between two values
    skip_factor -= 32;
    back_dis = skip_factor;
}
last_hit = hitnumber;
```

Figure 6.5. Adaptive update of the variable ‘skip_factor’.
Since our adaptive approach needs the L3 cache hit statistics for CPU, we introduce a new instruction to access this L3 cache performance counter and this new instruction is implemented through a new ‘ptlcall_cpu_read_value’ function in our simulator. As shown in Figure 6.5, if the CPU has too many L3 cache hits, we increase the ‘skip_factor’ by 32. If the CPU has too few L3 cache hits, the variable is decreased by 32. The check ‘if(back_dis != skip_factor -32)’ prevents ‘skip_factor’ from bouncing between its current value and (the current value – 32). The variable ‘back_dis’ is initialized as 0. As discussed in section 6.4.1 the constant value of 32 is used as ‘2 x L3 cache line size / the size of float’. In Figure 6.6, we examine the effectiveness of our proposed adaptive update approach. In this figure, we report the value of the variable ‘skip_factor’ over time for different benchmarks. We can see that for most benchmarks, the variable quickly converges to a fixed value, implying that the GPU kernel has a stable memory access pattern and the CPU pre-execution code keeps a fixed distance ahead of GPU threads. For the benchmarks BS and MV, the value of this variable changes over time, indicating that their memory access patterns are not stable due to the data dependent nature of sorting algorithms (BS) and the L1/L2 caching effects (MV).

![Figure 6.5](image)

**Figure 6.5.** The value of the variable ‘skip_factor’ over time using our adaptive update approach shown in Figure 6.5.

Since many GPU workloads have regular memory access patterns, as shown in Figure 6.6, we also propose to use profiling to simplify the update of the variable ‘skip_factor’. In the profiling process, the compiler sets the ‘skip_factor’ to a fixed value from the set {32, 64, 96, 128, 160, 192, 224} and
selects the one with highest performance during test runs. This way, the periodic update of ‘skip_factor’ can be removed from the CPU pre-execution code and there is also no need for a new instruction to access the L3 cache hits for the GPU. The CPU pre-execution code with a fixed ‘skip factor’ is shown in Figure 6.7, from which we can see that the code related to ‘batch_size’ is also removed.

6.4.3 Generating the CPU Pre-Execution Code from HWST GPU Kernels

HWST GPU kernels contain one or more loops, which contain global memory access instructions. We refer to such a loop as a kernel loop. Among our benchmarks, MM, Con, TMV, MV, and MC are of the HWST type. To generate the CPU pre-execution program for an HWST GPU Kernel, we process one kernel loop at a time. For each kernel loop, a CPU function is generated, which contains the global memory access instructions and the address computation operations in the loop body. Both the thread id and the kernel loop iterator are replaced with function input parameters. If the kernel loop is a nested one, the iterators from all loop levels are replaced with function parameters. Our proposed compiler algorithm is shown in Figure 6.8.

```c
float cpu_prefetching( ... ) {
    unroll_factor = 8; skip_factor = 160;
    // added loop to traverse thread blocks
    for (j = 0; j < N_tb; j+= concurrent_tb) {
        // added loop to traverse concurrent threads
        for (i = 0; i < concurrent_tb*tb_size ;
            i+=skip_factor*unroll_factor) {
            int thread_id=i+skip_factor*unroll_factor+j* tb_size;
            // unrolled loop
            float a0=memory_fetch_for_thread (
                thread_id+ skip_factor *0);
            float a1=memory_fetch_for_thread (
                thread_id+ skip_factor *1);
            ........
            sum += a0+a1+a2+a3+a4+a5+a6+a7; /* operation
                inserted to overcome dead code elimination */
        }}
}
```

Figure 6.7. The CPU pre-execution code for the vector-add GPU kernel with a fixed skip factor of 160.
As shown in Figure 6.8, after generating the function to load data for one loop iteration of a kernel loop (i.e., step 1), we insert loops to prefetch data for many concurrent threads. Similar to LWST kernels, the outer loop is used to prefetch data for concurrent TBs. Before going through concurrent threads, however, we insert the second-level loop to account for the kernel loop structure. The third-level loop traverses through all concurrent threads, similar to step 2b in Figure 6.2 for LWST kernels. We illustrate our algorithm using the simplified version of transpose-matrix-vector multiplication. The GPU kernel and the generated CPU pre-execution program are shown in Figure 6.9.

As shown in Figure 6.9, the CPU function ‘memory_fetch_for_thread_loop_1’ is generated from the loop body of the GPU kernel and the loop iterator and thread id are replaced with function parameters. In CPU function ‘cpu_prefetching’, the second-level loop (with iterator ‘m’) corresponds to the kernel loop. The iterator update is a fixed value 8 rather than 1 so as to unroll the loop body for 8 times. The third-level loop (with iterator ‘i’) traverses through concurrent threads for prefetching. The reason for such loop organization is that GPU executes many threads in parallel. Therefore, instead of prefetching data of multiple iterations for one thread, we prefetch data of one iteration for many threads before moving on to the next iteration. The ‘skip_factor’ update part in Figure 6.9 is the same as discussed in Section 6.4.2 and both adaptive and profiling approaches can be applied.

From our algorithms shown in Figures 6.2 and 6.8, we can see that the granularity of our CPU prefetch function is one loop iteration, with LWST as a special case of HWST. One may suggest finer

---

1. For each kernel loop, extract memory operations and the associated address computations from the loop body and put them in a CPU function, replace thread id computation with an input parameter, and replace the kernel loop iterators with input parameters.
2. Add a nested loop structure into the CPU code to prefetch data for concurrent threads.
   a. The outer loop traverse through all TBs. The iterator starts from 0 and the loop bound is the number of thread blocks in the GPU program. The iterator update is the number of concurrent TBs, meaning the number of TBs that can run concurrently on the GPU.
   b. The second-level loop corresponds to the kernel loop and we use the same loop bound. The loop update is increased to unroll the next level loop. If the kernel loop is nested, this second-level loop is also nested.
   c. The third-level loop traverses through concurrent threads. The loop iterator starts from 0 and the loop bound is the number of concurrent threads (which is the product of TB size and the number of concurrent TBs). The iterator update is set as a product of three parameters, unroll-factor, batch_size, and skip_factor.

Figure 6.8. The compiler algorithm to generate CPU pre-execution program from HWST GPU kernels.
granularity such as prefetching one memory access at a time. In other words, the CPU fetches one datum (e.g., A[n] in Figure 6.3) for many threads before moving on the next (B[n] in Figure 6.3) rather than fetching all the data in one iteration for a thread (A[n], B[n], and C[n] in Figure 6.3). We do not choose this approach since it requires the CPU and GPU to follow the exactly same access order and the GPU compiler is more likely to re-order the accesses within a loop body than to reorder accesses across different loops. Furthermore, using one CPU function call to issue one access incurs too much control overhead for CPU execution.

Note that the algorithms in Figures 6.2 and 6.8 assume that the TBs are dispatched to SMs in-order, which is the case based on our experiments on current discrete GPUs. If out-of-order TB dispatch is used, our scheme would require GPU to send the active TB ids to the CPU and the prefetching is done accordingly for those active blocks. In other words, we will replace the implicit block ids in the loop “for (j = 0; j < N_tb; j+= concurrent_tb)” (step 2a in Figure 6.2 and Figure 6.8) with explicit ones forwarded from the GPU. Our sequential dispatch assumption eliminates such GPU-to-CPU communication.
Figure 6.9. A code example for HWST. (a) A Transpose Matrix Vector Multiplication GPU kernel; (b) the (partial) CPU pre-execution program.
6.5 Experimental Results

6.5.1 Performance of CPU-Assisted GPGPU

After the CPU pre-execution program is generated, we let the CPU to execute this program right after the GPU kernel is launched. In our first experiment, we examine the performance improvements achieved with our proposed CPU-assisted execution. In Figure 6.10, we report the GPU performance using instruction per cycle (IPC) for each benchmark for three configurations, no CPU pre-execution (labeled ‘no-preex’), CPU pre-execution with adaptive update of skip factor (labeled ‘adaptive’), CPU pre-execution with a fixed skip factor determined from profiling (labeled ‘profiling’). Since our GPU has 4 SMs and each SM has 32 scalar SPs, the peak IPC is 128. We also include the GPU performance results for a perfect L3 cache (labeled ‘perfect L3) in Figure 6.10 for reference.

![Figure 6.10. GPU performance comparison among no-pre execution (no-preex), CPU pre execution with adaptive update of ‘skip_factor’ (adaptive) and CPU pre execution with a fixed ‘skip_factor’ determined from profiling (profiling)](image)

Form Figure 6.10, we can see that our proposed CPU pre-execution improves performance significantly, up to 113% (MC) and 21.4% on average with adaptive update of ‘skip_factor’ and up to 126% and 23.1% on average using a fixed ‘skip_factor’ determined from profiling. Among these benchmarks, BSc, VD and TMV are memory intensive and we achieve about 30% speedups. The high performance gains from MC are due to the fact that the GPU kernel (without CPU pre-execution) suffers from partition conflicts [67] while our pre-execution exploits the partition-level parallelism of
off-chip memory when it prefetches data across multiple TBs. As the GPU requests hit in L3 cache, they do not go to off-chip memory, thereby avoiding the partition conflicts. The speedups for BS, MV, FFT are from 4% to 11% due to their irregular address patterns and cache conflicts. There are no performance benefits for MM and Con because they are highly optimized and have good locality and data reuse in L1 and L2 cache of GPU, which makes the L3 cache not critical. Even with a perfect L3 cache, the performance gains are negligible for these two benchmarks.

Another observation from Figure 6.10 is that both adaptive update of ‘skip_factor’ and a fixed ‘skip_factor’ selected from profiling are effective in improve the GPGPU performance. The profiling approach is slightly better as the adaptive approach usually quickly converges to the optimal value, as shown in Figure 6.6.

6.5.2 The Efficacy of Data Prefetching using CPU Pre-execution

In this experiment, we examine the efficacy of data prefetching using CPU pre-execution. First, we evaluate the coverage of this prefetching scheme by examining the L3 cache hit rate for GPU accesses with and without CPU prefetching. The results are shown in Figure 6.11. The hit rates for GPU execution without CPU pre-execution is labeled ‘no-preex’ and GPU execution with CPU pre-execution using the adaptive update of ‘skip_factor’ is labeled ‘adaptive’. The results for CPU pre-execution using fixed ‘skip_factor’ are very close to adaptive update. From Figure 6.11, we can see that the L3 cache hit rates are highly improved by CPU pre-execution. On average, it improves from 12.9% to 39.2%. The L3 cache hit rate improvements for MM and Conv do not translate to performance gains as shown in Figure 6.10. The reason is that TLP and higher level of caches provide sufficient latency hiding for these benchmarks.
Another metric for data prefetching is accuracy and we evaluate it by computing the ratio of how many L3 misses generated from CPU pre-execution are actually accessed by GPU threads and the results are shown in Figure 6.12. It can be seen from the Figure 6.12 that our proposed CPU pre-execution has very high accuracy. On average, 98.6% data blocks loaded from the memory by CPU are accessed by the GPU threads.

Since our proposed CPU pre-execution needs to execute instructions to generate prefetching requests for GPU threads, one way to evaluate the overhead of our approach is to examine how many instructions the CPU needs to execute in order to achieve the performance gains. In Figure 6.13, we report the ratio of the number of instructions executed by the CPU over the number of instructions
executed by GPU for both adaptive update of ‘skip_factor’ (labeled ‘adaptive’) and fixed ‘skip_factor’ selected using profiling (labeled ‘profiling’). From Figure 6.13, we can see that the performance gains shown in Figure 6.10 are achieved with little instruction overhead. On average, our CPU assisted GPGPU using adaptive update of ‘skip_factor’ (fixed value of ‘skip_factor’) only executes 0.74% (0.69%) extra instructions to deliver the performance gains.

6.5.3 Understanding the Impact of GPU Architectures

In this experiment, we vary the following GPU architecture parameters to understand the impact on our CPU-assisted GPGPU, the GPU SP frequency, the off-chip memory frequency, and the number of SPs in an SM. First, we vary the GPU SP frequency from the default 480 MHz to 267 MHz and 800 MHz. The CPU frequency remains at 2.4 GHz and the DRAM bandwidth remains at 19.2GB/s. In Figure 6.14, we report the speedups that are achieved from CPU assisted execution for each SP frequency compared to no CPU pre-execution (labeled ‘sp267_speedup’, ‘sp480_speedup’, and ‘sp800_speedup’). All CPU pre-execution uses adaptive update of ‘skip_factor’ in the experiments in this section and the fixed ‘skip factor’ has slightly better performance gains. The results labeled ‘sp480_speedup’ are what reported in Figure 6.10.
From Figure 6.14, we can see that when SP frequency is reduced, the relative memory latency is also reduced. Therefore, CPU pre-execution provides less performance gains. On the other hand, when we increase SP frequency, these benchmarks show different trends. First, the benchmarks, VD, FFT, BSc, have higher performance gains as the memory latency becomes more significant. Secondly, for the benchmarks, TMV, MV, and MC, the impact is opposite and the reason is that the GPU SPs nearly double the rate of its memory requests, which enforce the CPU to skip more threads and to prefetch less data. For the benchmark BS, its baseline IPC is very high (close to 90) when SP frequency is 480 MHz, thereby limiting the pre-execution impact as shown in Figure 6.10. Overall, increasing the speed of SPs has less impact than decreasing the speed and CPU pre-execution is still effective for all these different SP speeds.

Next, we vary the off-chip memory frequency from the default 600 MHz to 300 MHz and 1200 MHz and the results are show in Figure 6.15. From Figure 6.15, we can see that when memory frequency is increased, the memory latency is reduced. Therefore, CPU pre-execution provides less performance gains. When we reduce memory frequency, these trends of these benchmarks are similar to increasing the SP frequency. For example, for BSc, VD and FFT, the CPU pre-execution shows better speedups when the memory frequency is reduced from 600 MHz to 300 MHz, because the memory latency dominates these three benchmarks. For TMV, MV, and MC, memory latency does not dominate the execution time. Therefore, the speedups of CPU pre-execution are reduced when the memory frequency is reduced from 600 MHz to 300 MHz. Overall, reducing the speed of memory has much
less impact than increasing the memory speed and our proposed CPU pre-execution is effective for all these different memory speeds.

![Figure 6.15. The speedups from CPU pre-execution for off-chip memory running at different frequencies and the normalized execution time without pre-execution.](image)

Then, we vary the number of SPs in an SM and keep the same (4) SMs in our GPU model. In Figure 6.16, we report the speedups that are achieved from CPU assisted execution for each SM configuration compared to no CPU pre-execution (labeled ‘w16_speedup’, ‘w32_speedup’, and ‘w64_speedup’). In our baseline GPU configuration, each SM has 32 SPs. From the figure, we can see that when the number of SPs is reduced in an SM (while keep the same number of SMs in the GPU), the GPU becomes more latency tolerant as each instruction in a warp will take more cycles to finish. Given the same application, reducing the number of SPs is equivalent to increasing TLP, thereby reducing the performance gains achieved from CPU pre-execution. On the other hand, increasing the number of SPs also increases the rate of their memory requests, similar to increasing the SP speed, which can also reduce the effectiveness of CPU pre-execution. Nevertheless, on average, for these three SM configurations, CPU pre-execution achieves 14.7%, 21.4%, and 12.4% performance improvement, respectively.
6.5.4 Sensitivity of the Parameters in CPU Pre-Execution Program

In this experiment, we study the sensitivity of the two parameters used in our CPU pre-execution program to update the variable ‘skip_factor’ (see Section 6.4.2). The first is the ‘batch_size’, which determines how often the skip_factor is updated. We vary this variable from 8, 16, and 32 and the GPU performance results are shown in Figure 6.17. As seen from the figure, although the batch size of 16 achieves the best performance, the performance difference for different batch sizes is limited, except for BSc, which prefers a large batch size.
In another experiment, we also change the threshold used to determine whether there are too many or too few L3 cache hit. We vary the threshold from 10 to 50 and the results are nearly identical, showing that our algorithm is not sensitive to this parameter.

6.5.5 Using CPU to Execute GPU Threads

In this experiment, we consider the option of using the CPU to directly execute some GPU threads to reduce the GPU workload. On the GPU side, the thread blocks are distributed to SMs based on the order of thread block id from small to large. On the CPU side, the CPU executes the thread blocks from the opposite direction, starting from the one with the largest thread block id. In our simulator, we implemented a special instruction for the CPU to get the largest thread block id issued in the GPU. This way, we ensure that there is no overlap workload between the CPU and the GPU. The speedups of such workload distribution between CPU and GPU over GPU-only execution are shown in Figure 6.18. From the figure, we can see that the performance gains of most of benchmarks are less than 2%. The main reason is the limited floating-point throughput of the CPU and the high overhead of CPU to access GPU memory partition. Among the workloads, the benchmark, VD, shows the highest (about 5%) speedup since it does not have many ALU operations to expose the ALU bottleneck of the CPU.

![Figure 6.18. The speedups of workload distribution between GPU and CPU over GPU-only execution.](image)

6.6 Related work

Although a key design philosophy of GPU is to use TLP to hide long memory access latency, the importance of GPU memory hierarchy has been widely recognized to achieve high performance
GPGPU. In [65], software prefetching is used to overlap memory access latency with computations. However, prefetching data into registers or shared memory increases the register pressure and may hurt the performance due to reduced TLP [82]. In [41], Lee et al. proposed many-thread aware GPU prefetching approaches for L1 cache. Besides leveraging the well-known stride access pattern, they revealed the interesting insight that when a workload is parallelized into many threads, each thread may be short and inter-thread/inter-warp prefetching is more effective than intra-thread/intra-warp prefetching. Compared to this work, our proposed CPU pre-execution does not rely on stride access patterns and provides both intra- and inter-warp prefetching. More importantly, all the previous works [47] on GPU prefetching do not fit well with fused architectures as both demand cache misses and prefetches compete for critical resource, such as L2 cache miss handling status registers (L2 MHSRs), on the GPU side while leaving the CPU side resource idle. Our proposed approach, in contrast, leverages such critical resources on CPU side for prefetches and keeps those on GPU side for demand misses, thereby achieving better resource utilization. We also implemented the per-PC stride prefetcher with enhanced warp id indexing [41] in our simulator, which shows a 5.24% speedup on average. Another way to utilize the idle CPU is to distribute part of the workload to the CPU [47]. However it doesn’t fit with fused architecture because of the resource competition between the CPU and the GPU.

To take advantage of fused architectures, it is proposed in [79] that the GPUs run prefetching algorithms to prefetch data for CPU programs. In comparison, our goal is to accelerate GPU programs and we believe it is a better fit to fused architectures since both GPU and CPU are used to do what they are good at: GPU for ALU/floating-point computations and CPU for flexible and accurate data prefetching.

Our proposed CPU-assisted GPGPU is also inspired from many works on CPU-based pre-execution [8][18][38][46][49][64][72][78][92][93], in which a pre-execution thread is used to provide data prefetching and/or accurate control flow to the main thread. The novelty of our work is to use a single CPU thread to prefetch data for many concurrent GPU threads and a simple yet effective way to control how far the pre-execution thread runs ahead.

### 6.7 Chapter Conclusion

In this chapter, we propose to collaboratively utilize CPU and GPU resources for GPGPU applications. In our scheme, the CPU runs ahead of GPU threads to prefetch the data into the shared
L3 cache for the GPU. Novel compiler algorithms are developed to automatically generate CPU pre-execution programs from GPU kernels. We also provide flexible mechanisms to control how far the CPU runs ahead of GPU threads. Our experimental results show that our proposed CPU pre-execution has very high prefetching accuracy and achieves significant performance gains at the cost of minor instruction overhead from the CPU side. Furthermore, our results show that the proposed scheme remains effective for different GPU configurations.
Chapter 7 Conclusion

In this dissertation, first we present a compiler framework to optimize GPGPU programs. By applying a set of novel compiler techniques including data unit vectorization, memory coalescing, thread merge, thread block merge, data prefetching and partition camping elimination, our compiler can generate the optimized code achieving very high performance, even superior to manually optimized programs.

Secondly, we develop Apricot, an optimizing compiler and productivity tool for many-core coprocessors such as MIC. The key features of Apricot include automatic transformation of legacy applications, the identification of profitable code regions and optimizations to minimize data movement. The performance evaluation on Knights Ferry shows that our compiler can efficiently generate and improve the MIC program.

Thirdly, we develop novel software and hardware solutions to multiplex shared memory to improve the GPGPU throughput. Our software approaches essentially allow multiple thread blocks to shared one copy of shared memory in the runtime to increase TLP, and our hardware solution introduces new instructions to allocate and free shared memory so as to mitigate the side-effect of share memory usage. Our experimental results show that our proposed schemes improve the performance significantly.

Fourthly, we propose to collaboratively utilize CPU and GPU resources on fused architecture. In our scheme, since the CPU and the GPU shared the same L3 cache and off-chip memory, we let the CPU run ahead of GPU threads and prefetch the data into the L3 cache for the GPU. We provide either profiling-based mechanism or adaptive mechanism to control how far the CPU runs ahead of GPU threads. Our experimental results show that our CPU pre-execution can achieve very high prefetching accuracy and improve the performance significantly.

Overall, while the many-core architectures can provide tremendous capability, the programming models of many-core architectures present application developers with the new challenges, such as the memory transfer between the CPU and the accelerated device, the inter-thread optimization, and the software-managed cache. However, all these challenges have not been addressed by vendors’ compilers. It motivates us to develop the high level source to source compilers placed on the vendors’
compilers to bridge the gap. On the other hand, current many-core architectures employ the relatively simple architecture design to achieve the balance between the performance and the energy consumption. Our study of shared memory multiplexing shows that small architectural improvement can significantly increase the throughput of GPU programs with minor hardware cost. Furthermore, using prefetching as a case study, we show that we can use collaborative execution of a CPU and a GPU, rather than modify the GPU architecture to accommodate different purposes.
REFERENCES


[53] NVIDIA CUDA Toolkit 4.0 CUBLAS Library, 2011

[54] NVIDIA CUDA Toolkit 4.0 CUFFT Library, 2011


[58] OpenCL. http://www.khronos.org/openc/


