ABSTRACT

LI, CHAO. Optimizing Memory Efficiency for Many-Core Architecture. (Under the direction of Dr. Huiyang Zhou).

Massively parallel, throughput-oriented processors such as graphics processing units (GPUs) leverage high thread-level parallelism to overlap long latency memory accesses with computation. On-chip memory resources, including register files, shared memory, and data caches which were designed to provide high-bandwidth low-latency data accesses, remain critical to application performance.

In this dissertation, we study these memory resources and propose optimizations to improve the memory efficiency for GPU architecture in a whole stack, from hardware architecture, compiler, to application-level algorithms. First, I will present our work on understanding the tradeoffs between software-managed vs. hardware-managed caches in GPUs. To manage on-chip caches, either software-managed or hardware-managed schemes can be employed. State-of-the-art accelerators, such as the NVIDIA Fermi or Kepler GPUs support both software-managed caches, aka. shared memory, and hardware-managed L1 data caches (D-caches). Shared memory and the L1 D-cache on a GPU utilize the same physical storage and their capacity can be configured at runtime. In this work, we present an in-depth study to reveal interesting and sometimes unexpected tradeoffs between shared memory and the hardware-managed L1 D-caches in GPU architecture.

Secondly, I will present our novel compiler scheme for the judicious utilization of on-chip memory resources on GPUs. To manage these intricate on-chip memory resources is non-trivial for application developers. Moreover, the varying on-chip resource across different GPU generations makes performance portability a daunting challenge. In this study, we propose compiler-driven automatic data placement scheme, to refine GPU programs by altering data placement among different on-chip resources to achieve both performance enhancement and performance portability.

Thirdly, I will present our study on novel cache optimization for GPU architecture. On-chip L1 D-caches are critical resources for providing high-bandwidth and low-latency data accesses. We observe that the memory access streams to L1 D-caches for many applications
contain a significant amount of requests with low reuse, which greatly reduce the cache efficacy. We propose an efficient locality monitoring mechanism to dynamically filter the access stream on cache insertion such that only the data with high reuse and short reuse distances are stored in the L1 D-cache. We propose a design that integrates the locality filtering functionality into the decoupled tag store of the current L1 D-cache through simple and cost-effective hardware extensions.

Finally, I will present our study on algorithm-level memory efficiency optimization for GPUs. We optimize the memory efficiency for accelerating deep Convolutional Neural Networks (CNNs), the state-of-the-art machine learning algorithm, on GPUs. Existing works on GPU-accelerated deep CNNs mainly focus on the computational efficiency of CNNs while the memory efficiency of CNNs have been largely overlooked. In this study, we look into the memory efficiency of various CNN layers on GPUs and reveal the performance implications from both data layouts and memory access patterns. Then we propose a set of methods to optimize memory efficiency for accelerating CNNs on GPUs. The experiment results demonstrate the effectiveness of our memory optimizations and their universal effects on different types of layers and various complete networks.
Optimizing Memory Efficiency For Many-Core Architecture

by
Chao Li

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
2016

APPROVED BY:

Huiyang Zhou
Committee Chair

Gregory Byrd

Xipeng Shen

James Tuck
DEDICATION

To my wife Jing Rao, my parents, brothers, teachers and friends.
BIOGRAPHY

Chao Li was born in Lianyungang, a beautiful city in the northeastern Jiangsu province in China. He received his bachelor degree on software engineering from DaLian University of Technology (DLUT) in 2009 and got his master degree in computer science and technology from Institute of Software, Chinese Academy of Sciences (ISCAS) in 2012.

He started his PhD in computer engineering in 2012 at North Carolina State University (NCSU). Under the supervision of Dr. Huiyang Zhou, he has worked on several research projects in computer architecture and programming with a focus on memory system optimizations for GPUs. He has done internships with PNNL and NEC Labs America while he was a student. He got his PhD degree in computer engineering from NCSU in 2016 and began his career as a senior engineer in Qualcomm. He is a member of ACM and IEEE.
ACKNOWLEDGMENTS

I would like to express my greatest gratitude for those who have helped me through my PhD period. First and foremost, I would like to thank my PhD advisor, Dr. Huiyang Zhou. Without his support and guidance, this dissertation would not have been possible. I would like to express my greatest gratitude to him for the tremendous time, effort and wisdom that he has invested in my graduate research. His insightfulness on research and responsible attitude towards life will be treasured by me for a long time.

I also would like to thank Dr. James Tuck, Dr. Xipeng Shen and Dr. Gregory Byrd for serving on my PhD advisory committee. I would like to show my appreciation to Dr. Yi Yang, Dr. Frank Mueller, Dr. Shuaiwen Song and Dr. Shengen Yan, Dr. Albert Sidelnik, Dr. Siva Hari, Michael Mantor for their great suggestions and collaborations on my research.

Many thanks go to my colleagues including Hongwen Dai, Xiangyang Guo, Qi Jia, Zhen Lin, Yuan Liu, Yi Yang, Ping Xiang and Saraubh Gupta for their helps during my study. I especially want to thank Yi Yang and Hongwen Dai for their valuable advices in my research. Also, I would like to express my thanks to Amro Awad and Sungkwan Ku for their friendship and generous help.

I would like to thank my wife Jing Rao, my father Dayong Li, my mother Xiuju Zhang, my brothers Jiling Li, Yuling Li so deeply for their continuous support, trust and love.

Finally, I would like to dedicate this dissertation to all who have helped me, supported me and loved me in my life.
# TABLE OF CONTENTS

| LIST OF TABLES | viii |
| LIST OF FIGURES | ix |

Chapter 1 ............................................................................................................. 1
1Introduction ........................................................................................................ 1
Chapter 2 ............................................................................................................... 4
2Understanding the Tradeoffs between Software-Managed vs. Hardware-Managed Caches in GPUs ................................................................. 4
  2.1Introduction .................................................................................................... 4
  2.2Background ...................................................................................................... 6
  2.3Tradeoffs between shared memory and L1 D-caches .................................... 7
    2.3.1Case Study I: Matrix Multiplication ...................................................... 7
    2.3.2Case Study II: Fast Fourier Transform (FFT) ........................................ 13
    2.3.3Case Study III: Matching Cube (MC) ................................................... 17
    2.3.4Case Study IV: Path Finder (PF) ............................................................ 19
  2.4Experiments .................................................................................................... 21
    2.4.1Experimental Methodology ...................................................................... 21
    2.4.2Experimental Results on GTX 480 and GTX680 GPUs ......................... 23
    2.4.3Experimental Results on GPGPUsim ...................................................... 24
    2.4.4The TLP Impact of the D-Cache Versions ............................................. 27
    2.4.5Impact on Energy Consumption ............................................................ 28
    2.4.6Summary .................................................................................................. 30
  2.5Related Works ................................................................................................. 30
  2.6Conclusions ..................................................................................................... 31
Chapter 3 .............................................................................................................. 32
3Automatic Data Placement into GPU On-Chip Memory Resources .................... 32
  3.1Introduction .................................................................................................... 32
  3.2Background and Motivation ........................................................................... 34
  3.3Automatic Data Placement into On-chip Memory Resources ....................... 37
LIST OF TABLES

Table 2.1. The GPGPUsim configuration ................................................................. 21
Table 2.2. The benchmarks used in experiments .................................................... 22
Table 3.3. A comparison of hardware characteristics across different GPU generations .................. 36
Table 3.2. Parameters used in experiments ............................................................ 51
Table 3.3. Benchmarks and their resource usage .................................................... 52
Table 3.4. The auto-tuning time on GTX 680 ........................................................ 56
Table 4.1. Application categorization based on cache bypassing impact. CNF: Cache Unfriendly, CI: Cache Insensitive, CF: Cache Friendly. IPC is normalized to the case of all taking L1 D-Path (N-IPC). The selected applications include Particular Filter (PTF), Srad2 (SD2), Needleman-Wunsch (NW), Single-Source Shortest Path (SSSP), LU Decomposition (LUD), Hotspot (HS), Barnes-Hut (BH), CFD Solver (CFD), Leukocyte (LFK), Gaussian Elimination (GS), Fourier Transformation (FFT), Myocyte (MYC), PathFinder (PF), Srad1 (SD1), Heartwall (HT), Matrix Multiplication (MM), B+Tree (BT), and Back Propagation (BP) ........................................................................................................................................ 66
Table 4.2. Baseline architecture configuration ........................................................ 77
Table 4.3. Configurations of our proposed Decoupled L1D  ........................................ 78
Table 5.1. The CNNs and their layers used in the experiments ................................. 101
Table 6.1. Baseline Architecture Configurations .................................................... 130
Table 6.2. Evaluated Benchmarks ........................................................................... 131
LIST OF FIGURES

Figure 2.1. Code segments for the matrix multiplication kernel. (a) The kernel explicitly uses shared memory to store tiles (i.e., the shared-memory version); (b) the kernel implicitly uses the L1 D-cache to store tiles (i.e., the D-cache version). ................................................................. 9
Figure 2.2. Memory-level parallelism of the matrix multiplication kernel. (a) The D-cache version, (b) the shared-memory version. ............................................................................................................. 11
Figure 2.3. The D-cache version of matrix multiplication with prefetching instructions to improve MLP. .............................................................................................................................. 13
Figure 2.4. The code for 1-D FFT. (a) The shared-memory version; (b) the D-cache version. ............................................................................................................................................... 15
Figure 2.5. The code for accessing the intermediate result array (sh_mem) in the benchmark FFT ............................................................................................................................................ 16
Figure 2.6. The code segment of the benchmark Matching Cube (MC). (a) The shared-memory version, (b) the D-cache version. .................................................................................................. 18
Figure 2.7. The code segment of the benchmark Path-Finder (PF). (a) The shared-memory version, (b) the D-cache version. .................................................................................................................. 20
Figure 2.8. Performance comparison between the shared-memory versions and the L1 D-cache versions on a GTX 480 GPU. ........................................................................................................ 23
Figure 2.9. Performance comparison between the shared-memory versions and the L1 D-cache versions on a GTX 680 GPU. ............................................................................................................. 24
Figure 2.10. Performance comparison between the shared-memory versions and the L1 D-cache versions on GPGPUsim. The following cache models are evaluated for D-cache versions: 6-way 48kB with a write evict (WE) policy, 6-way 48kB with a write-back write-allocate (WBWA) policy, fully-associative 48kB with a WBWA policy (WBWA+FA). ..... 25
Figure 2.11. Cache miss rates of the shared-memory versions and the L1 D-cache versions on GPGPUsim. The following cache models are evaluated for D-cache versions: 6-way 48kB with a write evict (WE) policy, 6-way 48kB with a write-back write-allocate (WBWA) policy, fully-associative 48kB with a WBWA policy (WBWA+FA). The Shared Memory versions use a 4-way 16kB L1 D-cache ........................................................................................................ 26
Figure 2.12. The execution time of D-cache versions with different numbers of concurrent TBs per each SM. The execution times are normalized to the execution time of the shared-memory versions ........................................................................................................................................ 28
Figure 2.13. Energy comparison between the shared-memory versions and the L1 D-cache versions on GPUWattch. The following cache models are evaluated for D-cache versions: 6-way 48kB with a write evict (WE) policy, 6-way 48kB with a write-back write-allocate (WBWA) policy, fully-associative 48kB with a WBWA policy (WBWA+FA). ........................................ 29
Figure 3.1. Data placements among three types of on-chip memory. ......................................................................................................................................................... 38
Figure 3.2. A code example of PathFinder. ................................................................................................................................................. 39
Figure 3.4. A code segment of the benchmark SRAD. (a) The global memory version, (b) The register version ............................................................................................................................. 44
Figure 3.5. The compiler algorithm to promote variables from shared memory to the register file/D-cache ....................................................................................................................................... 46
Figure 3.6. The compiler algorithm to promote shared or global memory to registers to be shared among threads................................................................. 48
Figure 3.7. Performance speedups achieved by automatic data placement for all benchmarks on three different GPUs. ........................................................................................................... 53
Figure 3.8. Auto-tuning of our automatic data-placement for all benchmarks on GTX680 (Performance normalized to original kernel). .......................................................................................... 54
Figure 3.9. The optimal parameter, the number of shared memory array to be promoted and the C-Factor, determined for different GPUs. ..................................................................................... 55
Figure 3.10. Performance speedup of optimized kernels in Marching Cubes with different input sizes................................................................................................................................. 57
Figure 4.1. Memory request handling in the baseline architecture............................................. 64
Figure 4.2. L1 (right) and L2 (left) level contention study by increasing their associativity and capacity. It clearly shows that the major performance bottleneck resides at the L1 D-cache level for CNF workloads.............................................................................................................. 68
Figure 4.3. Reuse breakdown for CNF applications. It shows the percentage of addresses in L1 D-cache have been referenced by \( m \) number of times. The values of \( m \) are shown in the legend.......................................................................................................................... 70
Figure 4.4. The reuse distance histograms of L1 D-cache access stream of the CNF applications. The reuse distance is computed at cache access level instead of cache structure level (e.g. the reuse distance for access pattern ABCA is 2). ................................................................. 71
Figure 4.5. Hit rates and data reuse characteristics of GPU workloads: (a) Hit rates of CNF and CF applications; (b) The percentage of data accesses with and without data reuse. ...... 71
Figure 4.6. Decoupled L1D cache design diagram........................................................................ 73
Figure 4.7. Operation flow chart of the Decoupled L1D............................................................... 74
Figure 4.8. Performance comparisons of various cache management and bypassing schemes for the CNF workloads ............................................................................................................. 80
Figure 4.9. Performance comparisons of various cache management and bypassing schemes for the CF workloads. ............................................................................................................. 82
Figure 4.10. The reuse distance histogram of L1 access stream of PTF (a case with long reuse distance shown in Figure 4) shaped by different designs. The reuse distance is computed at the cache access level instead of cache structure level (e.g., the reuse distance for access pattern ABCA is 2). ........................................................................................................ 83
Figure 4.12. Impact of cache replacement policies on performance of CNF workloads on GPUs. ...................................................................................................................... 84
Figure 4.10. Cache performance (hit rate) and memory traffic caused by bypassing in various designs................................................................................................................................. 86
Figure 4.13. Design space exploration for our proposed decoupled L1 D-cache for GPUs. The design choices match Table 4.3 .................................................................................................................... 89
Figure 4.14. The effectiveness of our design on the baseline L1 D-cache with different capacities. All data is normalized to a 16KB baseline cache.................................................. 90
Figure 4.15. Performance of our design with various warp scheduling policies, normalized to pure GTO performance.................................................................................................................. 91
Chapter 1

Introduction

Throughput-oriented architecture, such as graphic processing units (GPUs), has been widely used to accelerate many general-purpose computation workloads. Although general purpose computation on GPUs (GPGPU) achieves high throughput mainly by employing a large bundle of threads to overlap computations with long latency memory accesses, off-chip memory bandwidth and latency remain a performance as well as energy-efficiency bottleneck. Furthermore, the current trend of GPGPU evolution scales the computational throughput much faster than off-chip memory access bandwidth. For example, Nvidia GTX480 GPUs based on the FERMI architecture [3] have an arithmetic throughput of 1.35 TFLOPSs with the memory bandwidth of 178 GB/s. In comparison, GTX680 GPUs based on the KEPLER architecture [4] have an arithmetic throughput 3.09 TFLOPS (2.3X increase over GTX480) with the memory bandwidth of 192 GB/s (7.8% increase over GTX480). To alleviate the off-chip bandwidth bottleneck and reduce memory access latency, GPUs are equipped with a multiple-level on-chip memory hierarchy including register files, L1 data caches (D-cache), shared memory, and L2 caches. As expected, how to effectively utilize such on-chip memory resources has a significant impact on application performance. In our first work, we aim to provide insights to the following questions on utilizing the two on-chip caches: (a) is it worthwhile for application
developers to explicitly manage shared memory with the existence of the hardware managed L1 D-caches in GPUs? And (b) what are the main reasons for code utilizing shared memory to outperform code leveraging L1 D-caches (and vice versa)? To answer them, we present an in-depth study to reveal interesting and sometimes unexpected tradeoffs between shared memory and the hardware-managed L1 D-caches in GPU architecture.

Secondly, as observed in our first work, the trade-offs among these on-chip memory resources are complex and sometimes non-intuitive. Thus, it is non-trivial for application developers to explicitly manage these on-chip memory resources for high memory efficiency. More importantly, as on-chip resources have been changing significantly for different generations of GPUs, an optimized kernel upon one generation becomes suboptimal on another one. Thus performance portability becomes a daunting challenge for application developers. We tackle this problem with compiler-driven automatic data placement. We focus on programs that have already been reasonably optimized either manually by programmers or automatically by compiler tools. Our proposed compiler algorithms refine these programs by revising data placement across different types of GPU on-chip resources to achieve both performance enhancement and performance portability.

Thirdly, on-chip caches, especially L1 data caches (L1 D-caches) which were designed to improve the performance of irregular workloads without programming scratchpads or shared memory, remain critical to provide high-bandwidth low-latency data accesses. We observe that the memory access streams to L1 D-caches for many applications contain a significant amount of requests with low reuse, which greatly reduce the cache efficacy. Existing GPU cache management schemes are either based on conditional/reactive solutions or hit-rate based designs specifically developed for CPU last level caches, which can limit overall performance. To overcome these challenges, we propose an efficient locality monitoring mechanism to dynamically filter the access stream on cache insertion such that only the data with high reuse and short reuse distances are stored in the L1 D-cache. Specifically, we present a design that integrates locality filtering based on reuse characteristics of GPU workloads into the decoupled tag store of the existing L1 D-cache through simple and cost-effective hardware extensions.
Finally, with the detailed study on memory resources on GPU architecture, we present a study on optimizing the memory efficiency for GPU-accelerated applications. State-of-the-art machine learning algorithms such as deep convolutional neural networks have become the critical GPGPU applications. A number of GPU-based CNN libraries have been developed. While existing works mainly focus on the computational efficiency of CNNs, the memory efficiency of CNNs have been largely overlooked. As deep neural networks have intricate data structure, the performance implication of memory behavior on GPUs is not straightforward. Our study unveils that there are two aspects that have not been addressed yet pose non-trivial impact on memory efficiency and the overall CNN performance, including both data layouts and memory access patterns. We look into these memory issues and propose a set of methods to optimize memory efficiency for accelerating CNNs on GPUs.
Chapter 2

Understanding the Tradeoffs between Software-Managed vs. Hardware-Managed Caches in GPUs

2.1 Introduction

To manage on-chip caches effectively, either explicit software management or implicit hardware management schemes have been widely used in computer systems. While hardware-managed caches relieve the application developers of explicit data management, it is expected that software approaches may offer higher cache performance (i.e., hit rates) with the knowledge of data reuse patterns. State-of-art graphics processing units (GPUs), such as the NVIDIA GTX480 and GTX680 GPUs, include both software managed caches, aka. shared memory, and hardware managed L1 data caches (D-caches). An outstanding feature of these GPUs is that shared memory and L1 D-caches utilize the same physical resource and their capacities can be configured through APIs. As a result, GPUs provide an ideal platform to study the intriguing tradeoffs between hardware-managed caches and software-managed caches.

In this work, we aim to provide insights to the following questions: (a) is it worthwhile for application developers to explicitly manage shared memory with the existence of the hardware
managed L1 D-caches in GPUs? And (b) what are the main reasons for code utilizing shared memory to outperform code leveraging L1 D-caches (and vice versa)?

We start our journey with the well-known matrix multiplication algorithm. From an optimized kernel using shared memory, we remove shared memory arrays while ensuring that the same tiling optimization has been applied. As the tiles fit in the D-cache/shared memory capacity, both versions enjoy very high hit rates after the tiles are initially loaded into shared memory/the L1 D-cache. The measured execution times on GTX480 GPUs (GTX680 GPUs do not cache global memory data in L1 D-caches), however, show that the L1 D-cache version is surprisingly much slower (43.8%) than the shared-memory version. Puzzled with these results, we resort to a GPU architectural timing simulator, which reports a similar performance trend, to retrieve detailed execution statistics. Through micro-kernels, assembly-level code analysis, and cycle-by-cycle instruction execution information, we pinpoint the unexpected reason: the shared-memory version achieves much higher memory-level parallelism (MLP) than the L1 D-cache version. Other factors including hit rates, memory coalescing, and dynamic instruction counts, have relatively little impact in comparison.

Besides matrix multiplication, we also perform detailed case studies on FFT, MC, and PF due to their distinctive data access patterns. Then, we categorize a set of 16 GPGPU workloads based on whether or not there is data sharing among threads. Our results show that for most applications, the GPU kernels utilizing shared memory deliver significantly higher performance than those leveraging L1 D-caches. The fundamental reasons are MLP and coalescing. For a few benchmarks for which the L1 D-cache versions have higher performance, the performance impact is mainly due to improved thread-level parallelism (TLP) and allocating more data to registers. Overall, rather than cache hit rates, the subtle factors including MLP, coalescing, and TLP often have more profound performance impacts.

The remainder of the chapter is organized as follows. Section 2.2 presents a brief background on GPU computing with an emphasis on GPU memory hierarchy. Section 2.3 contains four detailed case studies to reveal the interesting tradeoffs between shared memory and L1 D-caches. Section 2.4 categorizes the benchmarks based on their data access patterns.
and discusses their tradeoffs in utilizing shared memory or L1 D-caches. Section 2.5 addresses the related work. Section 2.6 concludes the chapter.

### 2.2 Background

State-of-art GPUs leverage high degrees of TLP to achieve high computational throughput. In GPU programming models such as NVIDIA CUDA [6], a GPU kernel is launched as a grid of thread blocks (TBs) and each TB in turn contains many threads. During execution, the threads in a TB are grouped into multiple warps based on their thread identifiers (ids) and threads in a warp execute in a lockstep manner, meaning that there is one program counter (pc) for all the threads in a warp.

Although TLP is key to hide long memory-access latencies, the GPU memory hierarchy remains critical to many GPU applications. The GPU memory space consists of texture memory, constant memory, local memory, shared memory, and global memory. The texture and constant memory are for read-only data and are accessible to all the threads in different TBs. Global memory can be read and written to by all the threads. In contrast, local memory is private for each thread while shared memory contains data that are local to each thread block. Shared memory is physically located on-chip, providing low access latencies and high bandwidths. As shared-memory variables are explicitly defined and used in GPU programs, it is a software-managed cache, similar to the local store in the CellBE architecture and scratchpad memory used in embedded systems. In comparison, hardware-managed caches are used for different types of GPU memory. Besides dedicated read-only caches for texture and constant memory, recent GPUs, including the NVIDIA Fermi [3] and Kepler [4] architectures, provide a level-one (L1) D-cache in each streaming multiprocessor (SM). In the Fermi architecture, the L1 data cache is used for both local and global memory data. In Kepler architecture, the L1 data cache is used only for local memory data. To simplify the cache-coherence management, the L1 caches employ a write-evict policy for global memory data in the Fermi architecture. Besides the L1 D-caches, there is a unified L2 cache shared among multiple SMs. Due to the limited capacity of shared memory in each SM, one side effect of
shared-memory usage is that it may limit the number of TBs that can run concurrently in an SM.

2.3 Tradeoffs between shared memory and L1 D-caches

2.3.1 Case Study I: Matrix Multiplication

For matrix multiplication with large matrices, the key optimization is loop tiling/blocking to reduce the working set size and thereby reduce reuse distances. The matrix multiplication kernel from CUDA SDK [5] uses shared memory to store the tiles and the code is shown in Figure 2.1a.

The code in Figure 2.1a computes a tile of elements in the product matrix. It reads a tile from either of the two input matrices and stores the two tiles into shared memory. Then, it performs multiplication between the two tiles before moving on to the next set of tiles. Since the tiles already reside in shared memory when multiplication is performed, the accesses to the tiles have low latency, thereby achieving high performance. This code illustrates the explicit way of managing data, which dictates where the data are stored and where to access them. It also showcases an overhead of shared-memory usage. The data are loaded to registers first and stored to shared memory. The same data will then be loaded again from shared memory into registers to perform computation. Such redundant data accesses result in increased dynamic instruction counts.

When an L1 D-cache is used, the tiles are stored in cache implicitly. When one element is loaded for computation, its neighbors in the same cache block/cache line will be loaded into cache. It does not need additional instructions to do so. Subsequent accesses, when hitting in cache, will load data from the cache directly. The code for the matrix multiplication kernel using the L1-D-cache is shown in Figure 2.1b. From the Figure 2.1a and Figure 2.1b, we can see that both versions of code use exactly the same tiling technique. The difference between the two versions is that in the inner loop, the shared-memory version accesses the data from shared memory explicitly while the D-cache version accesses the data from global memory, therefore the cache implicitly. For the shared-memory version, we set the shared memory
capacity as 48kB on a GTX480 GPU. For the D-cache version, we configure the L1 D-cache capacity as 48kB on the same GPU. The input matrices have a size of 256x256 and the tile size is 16x16. Therefore, each tile has a size of 1kB (=16x16x4B). With either configuration, the register usage is the limiting factor on how many TBs can run concurrently on an SM. For both versions, 5 TBs can run concurrently on a TB. Since the L1 D-cache and shared memory actually use the same physical on-chip storage and the capacity is large enough for the working sets (10kB for 5TBs), our initial expectation is the two versions should have similar performance. If we consider the instruction overhead for explicit data movement, the shared-memory version may have a slightly lower performance. Our actual experimental results on the GTX480 GPU, however, show that the D-cache version, (i.e., the code in Figure 2.1b) is 43.8% slower than the shared-memory version (i.e., the code in Figure 2.1a).

To explain the unexpected experimental results, we first use a micro-benchmark, similar to the one used in [16], to measure the shared-memory access latency and the L1 D-cache hit latency. Our experimental results show that for GTX480, the shared-memory access latency is 44 cycles and the L1 D-cache hit latency is 80 cycles. However, considering the fact that the benchmark matrix multiplication has a high degree of TLP to hide such latencies, the latency difference may not be a reason for our observed performance difference. Also, it is not clear how much the data movement overhead costs on the actual hardware. Therefore, we resort to a microarchitectural-level timing simulator, GPGPUsim V3.2.1 [2], to simulate the two code versions on a GPU configuration similar to GTX480. The shared-memory access latency and the L1 D-cache hit latency are 3 cycles and 1 cycle, respectively, in this simulator shared memory and the L1 D-cache both have the same throughput of 1 access per cycle. The results from the simulator show a trend similar to our actual hardware measurements, i.e., the D-cache version is 112.2% slower than the shared-memory version.

We collected the dynamic instruction counts for both versions of code shown in Figure 1, indicating that the shared-memory version has 12.7% more instructions than the D-cache version. This confirms our initial analysis on the data movement overhead of the shared-
memory version but offers no explanation why the shared-memory version has much higher performance.

Suspecting that the D-cache may suffer from conflict misses since the capacity is sufficient for the working sets, we use a perfect cache without any misses. Our results show that even with a perfect cache, the shared-memory version is still 0.2% faster than the D-cache version. Puzzled with the results, we craft a microkernel to check whether it is possible for a D-cache version to outperform its equivalent shared-memory version. This microkernel loads a block of data and then re-accesses the same block many times. The results from this microbenchmark confirm that the D-cache version is 13.0% faster than the corresponding shared-memory version (i.e., the block of data is located in shared memory). The data movement in the shared-memory version accounts for 8.3% dynamic instruction count overhead in this

Figure 2.1 Code segments for the matrix multiplication kernel. (a) The kernel explicitly uses shared memory to store tiles (i.e., the shared-memory version); (b) the kernel implicitly uses the L1 D-cache to store tiles (i.e., the D-cache version).
microkernel. With this result, now we are facing two key questions for the matrix multiplication kernel: (a) Why is the D-cache version slightly slower than the shared-memory version even with a perfect cache? (b) With a realistic cache, why is the D-cache version much slower?

To answer the first question, we look into the cycle-by-cycle instruction execution through the GPU pipeline. We observe that the load instruction corresponding to the array access ‘A[a+WA*ty+k]’ in Figure 2.1b experiences additional pipeline stalls. The reason is that for a warp of 32 threads, when the TB dimension is 16x16, will have two rows of threads. For example, threads in warp 0 in TB 0 will have the thread id along the X direction, threadIdx.x, ranging from 0 to 15 and the thread id along the Y direction, threadIdx.y (i.e., ty in A[a+WA*ty+k]), ranging from 0 to 1. As a result, for this warp, the load instruction will access A[a+WA*0+k] and A[a+WA*1+k]. With a cache-line size of 128 bytes, these two accesses will fall into two separate cache lines. In other words, two cache accesses are needed to complete this load instruction for each warp using the D-cache version. In contrast, for the shared-memory version, the tiles are stored into shared memory before the loop. Then, inside the loop, the array access ‘AS(ty,k)’ will correspond to AS(0,k) and AS(1,k) for warp 0. With shared memory featuring 32 banks and the dimension of the array AS of 16x16, both the elements AS(0,k) (i.e., AS[k]) and AS(1,k) (i.e., AS[1x16+k]) are located in the same row with k ranging from 0 to 16. Therefore, one shared memory access completes the shared-memory load instruction corresponding to ‘AS(ty,k)’. Since the array access ‘A[a+WA*ty+k]’ of the D-cache version cannot be coalesced into a single cache access, it suffers from additional pipeline stalls even though all accesses hit in cache. Such overhead outweighs the benefit of fewer instruction counts compared to the shared-memory version, thereby resulting in lower performance even with a perfect cache.

To answer the second question, we first vary the cache associativity from the default value of 6 to fully associative. The resulting cache miss rate improves from 3.59 MPKI (misses per 1k instructions) to 1.79 MPKI, meaning that there are conflict misses due to limited set associativity. The performance is also improved by 12.8%. We also vary the capacity from
16kB to 48kB but the miss rate does not vary significantly. This is reasonable as the working set of an SM, i.e., 5TBs, is just 10kB. Therefore, considering the large performance gap, cache misses should not be the key culprit of the poor performance of the D-cache version. Again, we resort to cycle-by-cycle instruction-level analysis to understand what happens at run-time. We found that performance is not determined by the total number of cache misses. Instead it depends more on how these cache-misses overlap with each other, i.e., the degrees of memory-level parallelism (MLP).

Figure 1.2. Memory-level parallelism of the matrix multiplication kernel. (a) The D-cache version, (b) the shared-memory version.
With the D-cache version, for each warp, during each iteration of the inner loop ‘for (int k = 0; k < BLOCK_SIZE; ++k)’, the access to array B ‘B[b+k*WB+tx]’ is a cache miss (e.g., tx = 0~15 for warp 0, warp 1, etc.) due to the large value of ‘WB’. In other words, for each different k, the access ‘B[b+k*WB+tx]’ is a cache miss. The access to array A ‘A[a+WA*ty+k]’ results in two misses (e.g., ty=0~1 for warp 0, ty = 2~3 for warp 1, etc.) for the first iteration (i.e., k=0) and hits in the cache for subsequent iterations. These two loads are followed by a dependent floating-point multiply-and-add (fma) instruction. Therefore, the cache misses are distributed across iterations. Considering a TB with a round-robin warp scheduling policy, the warps make similar progress and different warps can also overlap their misses, as shown in Figure 2.2a. However, as the access to array B ‘B[b+k*WB+tx]’ has an index independent on the thread id along the Y direction, with a TB size of 16x16, all warps in a TB will access the same address during the kth loop iteration. In other words, the misses due to accessing array B remain distributed across loop iterations. In comparison, for the shared-memory version, the code before the inner loop forces the tiles to be loaded from global memory and stored in shared memory before computation. For array B, the access ‘B[b+WB*ty+tx]’ ensures that the entire tile from array B will be loaded by different warps: warp 0 accesses ‘B[b+WB*ty+tx]’ with tx=0~15 and ty=0~1; warp 1 accesses ‘B[b+WB*ty+tx]’ with tx=0~15 and ty=2~3, etc. Therefore, all the cache misses from different warps in a TB are aggregated and overlapped with each other, as shown in Figure 2.2b. Comparing Figure 2.2a with Figure 2.2b, we can see that much higher MLP is achieved with the shared-memory version. Therefore, with a realistic cache, the performance of shared memory is much better than the D-cache version although both versions have similar numbers of total cache misses.

One way to improve MLP for the D-cache version is to use prefetch instructions to aggregate the cache misses together, as shown in Figure 2.3. However, comparing the code shown in Figure 2.1a and Figure 2.3, we can see that they have similar complexity and since the subtle MLP behavior is not obvious, it would be non-intuitive to engage in such a prefetching optimization. Furthermore, even with prefetching instructions, the code in Figure 2.3 still has a lower performance (8%) than the shared-memory version in Figure 2.1a due to non-coalesced cache accesses.
In summary, for matrix multiplication, the two key reasons for its shared-memory version to outperform the D-cache version are (a) its high MLP as the tiles are loaded into shared memory before computation; and (b) shared memory has 32 banks and is much less susceptible to the coalescing problem compared to the D-cache. In comparison, the classical metric of cache performance, i.e., the cache-miss/hit rate, is less critical than these two factors.

### 2.3.2 Case Study II: Fast Fourier Transform (FFT)

In this case study, we use a radix-4 based 1-D FFT kernel. The test input is a 1k-point FFT with a batch size of 4k. For a 1k-point FFT, each thread goes through multiple stages with each stage performing a 4-point FFT and then exchanging data with other threads. As a result, the key to achieve high throughput is to hold all the intermediate results in shared memory or the D-cache. The sample code of the radix-4 based FFT using shared memory and the D-cache is shown in Figure 2.4. Comparing the two versions, we can see that they are identical except that one uses a shared-memory array to keep the intermediate results and the other uses a global-memory array to do so. If the cache is large enough to keep the temporary data, all the accesses will be cache hits and enjoy low access latency.

For the two FFT kernels, we first test them on a GTX480 GPU. For the shared-memory version, the shared-memory capacity (48kB) limits the number of concurrent TBs on an SM to
be 5. For fair comparison, we also apply the same limit to the D-cache version with an unused share-memory array in the code. We also vary the TB parameter in our experiments in Section IV. Our experimental results show that the shared-memory version is 3.77X faster than the D-cache version.

The first reason for this high performance disparity is that on the GTX480, the L1 cache adopts a write-evict (WE) policy. As seen from Figure 2.4b, after each 4-point FFT computation, the results are written to the global-memory array, meaning that the data will be evicted from the cache. Therefore, the L1 cache provides no benefits for subsequent reads. To further analyze the impact of the cache-write policy, the microarchitectural-level simulator, GPGPU\textit{sim}, is used again. With a WE policy, it reports that the D-cache version is 119% slower than the shared-memory version. When we change the write policy to write-back and write-allocate (WBWA), the performance of the D-cache version improves by 26% but is still much slower than the shared-memory version.

To determine the reason, we then change the L1 D-cache to be fully associative. Compared to the default 6-way set associative configuration, both the total number of cache misses and the overall execution time changes very little (less than 1%). We also increase the cache capacity to 64kB, 128 kB, and 1MB and there is little performance impact. Therefore, cache contention and cache capacity are not the reason why the D-cache version under-performs the shared-memory version. Also, unlike matrix multiplication, where the tiles are loaded to shared memory prior to computation, for FFT, both the shared-memory version and the D-cache version load the global-memory data and access the intermediate results in the same way, as shown in Figure 2.4. So, there is not much difference in their MLP either.
Compared to shared memory, the L1 D-cache with a write-allocate policy may suffer a penalty due to write misses, which is the case for FFT when the intermediate result array is updated after the first stage of a 4-point FFT. If only a part of a cache line is to be updated, the full cache line will need to be fetched first from the off-chip memory, incurring a high penalty. To analyze such impact, we change the simulator such that the first write misses to the intermediate result array are treated as hits. With such a change, the performance of the D-cache version is improved by 21%.

Another key factor that affects the performance of the D-cache version is the un-coalesced cache accesses. As shown in Figure 2.5, the 1-D shared memory array 'sh_mem' is used for intermediate results for the shared-memory version. During one 4-point FFT pass of the 1k-point FFT computation, the intermediate results are stored to the ‘sh_mem’ array using the
function ‘SavetoSm’. The array index or offset is computed with ‘(tid<<2)+(tid>>3)’ where ‘tid’ is generated from the thread ids as shown in Figure 2.5. Such an index computation eliminates the bank conflicts in shared memory accesses and each shared-memory load instruction from a warp can be satisfied with a single shared-memory read transaction. For example, for warp 0 in TB0, its ‘tid’ ranges from 0 to 31. With ‘(tid<<2)+(tid>>3)’, the mapped indices become 0, 4, 8, 12, 16, 20, 24, 28, 33, 37, 41, 45, etc. Considering that there are 32 banks in shared memory, different threads in a warp will access a different bank in shared memory. However, when we replace this shared-memory array with a global memory array, the L1 D-cache cannot leverage such interleaved indices. These indices cannot be coalesced to a single cache line access and instead have to be mapped to 4 different cache lines with a cache line size of 128 bytes. As a result, each array update (i.e., a store-to-global-memory instruction with such indices/offsets) will require 4 updates to the L1 D-cache. The same access patterns are also present in the load operations. Considering the L1 D-cache/shared memory has one read port and one write port, each L1 D-cache access incurs significantly higher latency. Therefore, the D-cache version achieves much lower performance than the shared-memory version.

```c
#define   SavetoSm(distance, stide_xyzw, offset) { \ 
  float *pv_shm=sh_mem+offset; \ 
  for(int i=0;i<4;i++) { \ 
    pv_shm[0*stride_xyzw] = zr[i].x; \ 
    pv_shm[1*stride_xyzw] = zr[i].y; \ 
    pv_shm[2*stride_xyzw] = zr[i].z; \ 
    pv_shm[3*stride_xyzw] = zr[i].w; \ 
    Pv_shm+=distance; \ 
  } \ 
} __global__ void fft1k(float * greal, float * gimg) { \ 
  __shared__ float sh_mem[68*4*4*2]; \ 
  uint gid = threadIdx.x+blockIdx.x*blockDim.x; \ 
  uint tid = gid & 0x3fU; \ 
  ... \ 
  FFT4(); \ 
  SavetoSm(66*4, 1, (tid<<2)+(tid>>3)); \ 
  ...}
```

Figure 2.5. The code for accessing the intermediate result array (sh_mem) in the benchmark FFT.
In summary, for the benchmark FFT, three key factors affect the performance of the D-cache version. First, the write-evict policy makes the cache useless as there are updates to the intermediate result array before it is read again. Second, even with a write-back write-allocate L1 D-cache, the first write misses incur non-trivial performance penalty. Third, the un-coalesced cache accesses significantly increase the latency of each load/store from/to the intermediate result array.

2.3.3 Case Study III: Matching Cube (MC)

The benchmark MC has multiple kernels and the kernel ‘generateTriangles’ dominates the performance. For this kernel, each TB has 32 threads and the launched grid contains 1024 TBs. The code segment of this kernel is shown in Figure 2.6a. It uses two shared memory arrays, ‘vertlist’ and ‘normlist’ for its intermediate results. The indices used to access these two arrays, as shown in Figure 2.6a, are ‘threadIdx.x + i*NTHREADS’, where ‘NTHREADS’ is 32 and ‘i’ ranges from 0 to 11. As a result, the data stored in these two shared memory arrays are actually not shared among different threads in a TB. Instead, each thread uses a part of each array for its own intermediate data. Therefore, we can replace these two shared-memory arrays with local-memory arrays, which are private to each thread, and use the L1 D-cache to buffer such local memory arrays. The code utilizing the local-memory arrays is shown in Figure 2.6b. The main reason why the local-memory arrays are used rather than global-memory arrays is that the L1 D-cache in the Fermi architecture uses a write-back write-not-allocate policy for local memory data.

Given the size of the two shared memory arrays (2*4*3*12*32 = 9216 Bytes) in the shared-memory version, each SM can run 5 TBs concurrently when its shared memory size is configured to 48kB. For the D-cache version, as it does not use shared memory at all, each SM can run up to 8 TBs concurrently, where 8 is the maximal number of TBs to run on an SM. We test the two versions of the MC code on the GTX480 GPU. The results show that the D-cache version has 17.3% higher performance than the shared-memory version due to the higher number of concurrent TBs. If we use a fake shared memory array to set the number of concurrent TBs for the D-cache version to be 5, the D-cache version is 5.8% slower than the
shared-memory version due to the write-back write-not-allocate policy. Our experiments on GPGPU\textvisiblespace{\textregistered}Sim also report similar performance trends.

In summary, for the benchmark MC, the non-sharing usage of its shared-memory arrays enables them to be replaced with local-memory arrays. The key reason why the D-cache

__global__ void generateTriangles(...) {
    ...
    __shared__ float3 vertlist[12*\text{NTHREADS}]; // \text{NTHREADS} = 32
    __shared__ float3 normlist[12*\text{NTHREADS}];
    // defines to the shared memory array
    vertexInterp2(isoValue, v[0], v[1], field[0], field[1],
                  vertlist[threadIdx.x], normlist[threadIdx.x]);
    vertexInterp2(isoValue, v[1], v[2], field[1], field[2],
                  vertlist[threadIdx.x+N\text{THREADS}],
                  normlist[threadIdx.x+N\text{THREADS}]);
    vertexInterp2(isoValue, v[2], v[3], field[2], field[3],
                  vertlist[threadIdx.x+(N\text{THREADS}*2)],
                  normlist[threadIdx.x+(N\text{THREADS}*2)]);
    ...
    // uses of the shared memory array
    pos[index] = make_float4(vertlist[(edge*N\text{THREADS})+threadIdx.x], 1.0f);
    ...
}

__global__ void generateTriangles(...) {
    ...
    float3 vertlist[12];
    float3 normlist[12];
    // defines to the local memory array
    vertexInterp2(isoValue, v[0], v[1], field[0], field[1], vertlist[0],
                  normlist[0]);
    vertexInterp2(isoValue, v[1], v[2], field[1], field[2], vertlist[1],
                  normlist[1]);
    vertexInterp2(isoValue, v[2], v[3], field[2], field[3], vertlist[2],
                  normlist[2]);
    ...
    // uses of the shared memory array
    pos[index] = make_float4(vertlist[edge], 1.0f);
    ...
}

Figure 2.6. The code segment of the benchmark Matching Cube (MC). (a) The shared-memory version, (b) the D-cache version.
version outperforms the shared-memory version is that the D-cache version eliminates the shared-memory usage, which in turn removes the resource limitation on the number of concurrent TBs on each SM. Such improved TLP leads to higher performance for MC.

2.3.4 Case Study IV: Path Finder (PF)

The kernel code of the benchmark Path-Finder makes use of two shared memory arrays, ‘prev’ and ‘result’, as shown in Figure 2.7a. Its TB dimension is 256x1 and its thread grid size is 19x1. As a result, the sizes of these two shared-memory arrays are small (256x4=1kB) and such shared memory usage is not a bottleneck for the number of concurrent TBs on each SM. For the shared-memory array ‘prev’, its accesses in the kernel code, ‘prev[tx-1]’ and ‘prev[tx+1]’ indicate that the data in this array are shared among different threads. In contrast, the shared-memory array ‘result’ is always accessed with ‘result[tx]’, which means that there is no data sharing for this array. Therefore, we can simply use a register variable to replace it, as shown in Figure 2.7b. Further, as the variable is only defined and used in the same thread, we can safely remove the synchronization instruction ‘__syncthread()’ after the statement defining the variable ‘result’, as shown in Figure 2.7b. The same argument can also be made to remove the ‘__syncthread()’ instructions after defining the ‘result’ array element, as shown in Figure 2.7a.

As register accesses have the lowest latency and the shared-memory accesses also involve additional instructions to compute the addresses, the D-cache version shown in Figure 2.7b outperforms the shared-memory version shown in Figure 2.7a by 11.2%. If we do not remove the ‘__syncthread()’ instruction from the D-cache version in Figure 2.7b, it still outperforms the shared-memory version by 6.5%, highlighting the performance benefits by using registers to replace the shared-memory array ‘result’. In comparison, if we remove the ‘__syncthread()’ instruction from the shared-memory version, its performance improves by 4.7%, which still has lower performance than the D-cache versions. Using GPGPUsim, we find that by using registers to replace the shared-memory array, the D-cache version has 3.6% fewer dynamic instructions than the shared-memory version. When the ‘__syncthreads()’ instruction is removed, the D-cache version has 5.4% fewer instructions than the shared-memory version.
However, if we replace the prev array in shared memory with a global-memory array, the performance would degrade by 15.7%. This is mainly because re-accessing the frequently updated prev array in each loop hurts the cache performance with the WE policy.

Figure 2.7 The code segment of the benchmark Path-Finder (PF). (a) The shared-memory version, (b) the D-cache version.

In summary, for the benchmark PF, as one of its shared-memory array is not used for data communication across different threads, this shared-memory array can be replaced with register variables. The register variables have the lowest access latency and it also takes fewer
instructions to access registers than accessing shared-memory variables. Both lower latency and fewer dynamic instructions make the D-cache version outperform the shared-memory version for PF.

2.4 Experiments

2.4.1 Experimental Methodology

<table>
<thead>
<tr>
<th>Table 2.1. The GPGPUsim configuration.</th>
</tr>
</thead>
<tbody>
<tr>
<td># of execution cores</td>
</tr>
<tr>
<td>SIMD Pipeline Width</td>
</tr>
<tr>
<td>Number of Threads/Core</td>
</tr>
<tr>
<td>Number of Registers/Core</td>
</tr>
<tr>
<td>Shared Memory /Core</td>
</tr>
<tr>
<td>L1 Data cache/Core</td>
</tr>
<tr>
<td>MSHR Entry</td>
</tr>
<tr>
<td>Constant Cache Size /Core</td>
</tr>
<tr>
<td>Texture Cache Size/Core</td>
</tr>
<tr>
<td>L2 Data cache</td>
</tr>
<tr>
<td>Number of Memory Channels</td>
</tr>
<tr>
<td>Memory Channel Bandwidth</td>
</tr>
<tr>
<td>DRAM clock</td>
</tr>
<tr>
<td>DRAM Schedule Queue Size</td>
</tr>
<tr>
<td>Warp Scheduling Policy</td>
</tr>
</tbody>
</table>

In our experiments, we use both real GPU hardware and a microarchitectural timing simulator, GPGPUsim V3.2.1. Our experiments on the GTX480 GPU are performed on a Linux workstation with CUDA 4.0. We also perform experiments on a GTX680 GPU with CUDA 5.0. In our experiments using GPGPUsim V3.2.1, we use the configuration shown in Table 2.1, which is similar to the Fermi architecture, i.e., GTX 480 GPUs. As the L1 D-cache and shared memory utilize the same 64kB storage, we use 48kB L1 D-cache + 16kB shared
memory for the L1 D-cache versions and 16kB L1 D-cache + 48kB shared memory for the shared-memory versions.

### Table 2.2 The benchmarks used in experiments.

<table>
<thead>
<tr>
<th>Name</th>
<th>Threads</th>
<th>Blocks</th>
<th>Input size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Multiplication(MM)</td>
<td>(16,16)</td>
<td>(16,16)</td>
<td>256*256</td>
</tr>
<tr>
<td>Histogram(HG)</td>
<td>(64,1)</td>
<td>(128,1)</td>
<td>1024*1024</td>
</tr>
<tr>
<td>Matrix Transpose(MT)</td>
<td>(16,16)</td>
<td>(64,64)</td>
<td>1024x1024</td>
</tr>
<tr>
<td>Fast Walsh Transform(FWT)</td>
<td>(512,1)</td>
<td>(4096,1)</td>
<td>8388608</td>
</tr>
<tr>
<td>3D Laplace Solver(LPS)</td>
<td>(32,4)</td>
<td>(4,25)</td>
<td>100<em>100</em>100</td>
</tr>
<tr>
<td>Needleman-Wunsch(NW)</td>
<td>(16,1)</td>
<td>(64,1)</td>
<td>1024<em>1024</em>10</td>
</tr>
<tr>
<td>Back Propagation(BP)</td>
<td>(16,16)</td>
<td>(16,16)</td>
<td>65536</td>
</tr>
<tr>
<td>scalar product of vectors(SP)</td>
<td>(64,1)</td>
<td>(256,1)</td>
<td>8192*512</td>
</tr>
<tr>
<td>Fast Fourier Transform(FFT)</td>
<td>(64,1)</td>
<td>(4096,1)</td>
<td>4k*1k</td>
</tr>
<tr>
<td>Discrete Wavelet Transform(DWT)</td>
<td>(512,1)</td>
<td>(256,1)</td>
<td>256k</td>
</tr>
<tr>
<td>Image Blur(Blur)</td>
<td>(256,1)</td>
<td>(2,512)</td>
<td>1k*1k</td>
</tr>
<tr>
<td>StoreGPU(STO)</td>
<td>(128,1)</td>
<td>(384,1)</td>
<td>196656</td>
</tr>
<tr>
<td>Matrix-Vector Multiplication(MV)</td>
<td>(32,1)</td>
<td>(64,1)</td>
<td>2048*2048</td>
</tr>
<tr>
<td>MarchingCubes(MC)</td>
<td>(32,1)</td>
<td>(1024,1)</td>
<td>32768</td>
</tr>
<tr>
<td>N-Queue Solver(NQU)</td>
<td>(96,1)</td>
<td>(256,1)</td>
<td>8</td>
</tr>
<tr>
<td>Path Finder(PF)</td>
<td>(256,1)</td>
<td>(19,1)</td>
<td>4096<em>100</em>20</td>
</tr>
</tbody>
</table>

In this study, in order to cover the most common ways to utilize shared memory, we select 15 benchmarks that have shared memory usage from the NVIDIA CUDA SDK [12], AMD OPENCL SDK [1], GPGPU sim suite [2], Rodinia [4], and Opencv [1]. We also include a tiled version of matrix-vector multiplication [18] with a tile size of 32x32. The input and the kernel invocation parameters of these applications are shown in Table 2.2. Based on the way that shared memory data are reused, we classify the benchmarks into two categories. The first category uses shared memory for inter-thread communication and its data reuses are across threads. For applications in this category, we use global memory arrays to replace shared memory arrays, if needed. In the second category, applications use shared memory for intra-thread data reuse and we use local memory arrays or registers to replace shared memory.
variables. The second category includes the benchmarks STO, MV, MC, NQU, and PF. The remaining applications fall into the first category.

For a benchmark, if its shared-memory version limits the number of concurrent TBs on each SM, its corresponding D-cache version will have no such a constraint. In such cases, we allocate an unused shared-memory array to control the number of concurrent TBs to run on each SM. Then, we pick the best performing TB number for the D-cache version of the benchmark. More details on the TB number are discussed in Section 2.4.4.

2.4.2 Experimental Results on GTX 480 and GTX680 GPUs

We first present a performance comparison between the shared-memory versions and the D-cache versions on the GTX 480 GPU. As shown in Figure 2.8, the execution time of the D-cache version of each benchmark is normalized to the execution time of its corresponding shared-memory version. From the figure, we can see that on GTX 480, among the 16 benchmarks in our study, 14 benchmarks favor shared memory. Only two benchmarks, PF and MC, have higher performance with L1 D-cache versions than their shared-memory versions for the reasons discussed in Section III. On average, using the geometric mean (GM), the shared-memory versions result in 55.7% higher performance than the L1 D-cache versions.

![Figure 2.8. Performance comparison between the shared-memory versions and the L1 D-cache versions on a GTX 480 GPU.](image-url)
A performance comparison for the GTX 680 is presented in Figure 2.9. Compared to the Fermi architecture (i.e., GTX480), the Kepler architecture (i.e., GTX680) has more ALUs in each SM. Also, the L1 D-cache/shared memory in the Kepler architecture has a lower hit latency (33/33 cycles) than in the Fermi architecture (80/44 cycles). Unlike the Fermi architecture, global memory data are not cached in the L1 D-cache in the Kepler architecture. Despite these architectural differences, among the 16 benchmarks, only these same two benchmarks, PF and MC, achieve higher performance using the L1 D-cache version than the shared memory version.

### 2.4.3 Experimental Results on GPGPUsim

The performance comparison between the shared-memory versions and the L1 D-cache versions on GPGPUsim is presented in Figure 2.10. The performance for the L1 D-cache versions using both a write-evict (WE) policy and a write-back write-allocate (WBWA) policy are normalized to the performance of the shared memory versions. We also include the results of a fully associative L1 D-cache (FA+WBWA).
Compared to the actual hardware, the L1 D-cache has a 1-cycle hit latency and shared memory has a 3-cycle access latency in our GPGPUsim model. As a result, some benchmarks, such as FFT, NQU, and Blur, show smaller performance gaps between the L1 D-cache versions and the shared memory versions than the GTX480/680 results. On the other hand, for the benchmark MV, GPGPUsim reports much higher performance gaps between the L1 D-cache versions and the shared memory versions than the GTX480/680 results. The reason is that the L1 D-cache in GPGPUsim uses a basic index function, which leads to high numbers of conflict misses for MV.

Figure 2.10 Performance comparison between the shared-memory versions and the L1 D-cache versions on GPGPUsim. The following cache models are evaluated for D-cache versions: 6-way 48kB with a write evict (WE) policy, 6-way 48kB with a write-back write-allocate (WBWA) policy, fully-associative 48kB with a WBWA policy (WBWA+FA).

Several interesting observations can be made from Figure 2.10. First, for most benchmarks, the shared memory versions have higher performance than the D-cache versions, which is consistent with the real hardware results shown in Figure 2.8 and Figure 2.9.
Second, the WE policy hurts the cache performance if the updated data will be re-accessed again soon. This is the case for the benchmark FFT, as discussed in Section III-B. The benchmarks HG, MV, FWT, LPS, DWT, and BP also show similar behavior. Therefore, a WBWA policy can significantly improve the performance for these benchmarks. However, if the updated data will not be re-accessed, the WBWA policy may introduce cache pollution, thereby increasing the cache miss rate. The benchmarks STO, MT, and MC exhibit such a problem with the WBWA policy, as confirmed in the cache miss results presented in Figure 11.

Third, increasing set associativity reduces conflict misses and the benchmark MV drastically benefits from it. With the tiling optimization, the working set size of MV is within the cache capacity. However, its threads in a warp access the input matrix in a column-by-column manner. Therefore, given an input-matrix size of 2048x2048, different columns have

---

**Figure 2.11.** Cache miss rates of the shared-memory versions and the L1 D-cache versions on GPGPUsim. The following cache models are evaluated for D-cache versions: 6-way 48kB with a write evict (WE) policy, 6-way 48kB with a write-back write-allocate (WBWA) policy, fully-associative 48kB with a WBWA policy (WBWA+FA). The Shared Memory versions use a 4-way 16kB L1 D-cache.
the same cache line indices, resulting in a high number of conflict misses. A fully associative cache eliminates the problem, as also confirmed in Figure 2.11.

Fourth, with a WE policy, the D-cache versions outperform the shared-memory versions only for the benchmarks PF and MC, which is also consistent with the actual hardware results. With a fully associative cache and a WBWA policy, the benchmarks MV, NQU, and PF show higher performance using the D-cache versions than the shared-memory versions. For MV, since there is no data communication among the threads, we use registers to buffer each tile of the input matrix. As discussed before, register accesses have a lower latency and require a smaller number of instructions than shared-memory accesses. Furthermore, using registers eliminates the need for coalescing to perform column-based accesses. Therefore, the D-cache version using a fully associative cache for MV outperforms the corresponding shared-memory version. As discussed in Section III-C, the D-cache version for the benchmark MC benefits from increased TLP, thereby resulting in higher performance than its shared-memory version. The benchmark NQU also shows similar behavior to MC.

In Figure 2.11, we compare the cache performance, measured as misses per kilo instructions (MPKI), between the shared-memory versions and the D-cache versions. Note that the shared-memory versions still utilize the L1 D-cache when they access global memory or local memory. Also, for the shared-memory versions, the L1 D-cache size is 16kB. From Figure 2.11, we can see that the shared-memory versions typically have low cache miss rates, indicating that most data accesses are satisfied with shared memory. For the D-cache versions, a WBWA policy reduces the misses caused by write-evictions but may hurt some benchmarks as discussed before. From Figure 2.10 and Figure 2.11, we can also see that TLP significantly mitigates the performance impact of cache misses.

### 2.4.4 The TLP Impact of the D-Cache Versions

Among the benchmarks in our study, the shared-memory usage for the benchmarks FFT, MC, NQU, HG, and STO imposes a limitation on the number of TBs that can run concurrently on an SM. When the shared-memory usage is replaced, such a limit is eliminated and more TBs can be dispatched to an SM. We explore the number of TBs for the D-cache versions and
the performance results measured on the GTX 480 GPU are shown in Figure 2.12. The results on the GTX680 GPU and GPGPUsim show similar trends. Among these benchmarks, the maximal number of concurrent TBs per SM is 4 for the benchmark STO due to its register usage. For the remaining four benchmarks, up to 8 TBs can be dispatched to an SM, where 8 is a hardware limit [3]. From Figure 2.12, we can see that the benchmarks MC and STO prefer more TBs or higher TLP. For NQU, the TLP impact is relatively small as long as there are at least 3 TBs in each SM. For benchmarks FFT and HG, the best performance is achieved with 4 TBs and 6 TBs per SM, respectively. Further increasing the number of TBs per SM will result in cache contention, thereby hurting performance.

![Figure 2.12 The execution time of D-cache versions with different numbers of concurrent TBs per each SM. The execution times are normalized to the execution time of the shared-memory versions.](image)

2.4.5 Impact on Energy Consumption

We use GPUWatch [51] to analyze the impact of using shared memory vs. L1 D-cache on energy consumption and the results are in Figure 2.13. Similar to our performance comparison, we include the energy consumption results for L1 D-cache versions using a write-evict (WE) policy, a write-back write-allocate (WBWA) policy, and a fully associative write-back write
allocate L1 D-cache (FA+WBWA). The energy of D-cache versions is normalized to the energy of the shared memory versions.

As shown in Figure 2.13, the trend on energy matches the performance trend shown in Figure 2.10 for most benchmarks due to the significant static energy consumption modeled in GPUWattch and the lack of speculative execution in GPUs. One interesting exception is the D-cache version for the benchmark MC. The D-cache version enables more concurrent thread blocks (TB) and has lower execution time than its shared memory version. However, the high number of TBs leads to cache contention and results in higher dynamic energy consumption in memory hierarchy. Therefore, for MC, the D-cache version, although with lower execution time, consumes more energy than the shared-memory version.

Overall, for most of the benchmarks, the shared memory version consumes less energy than the D-cache version. On average, the shared memory versions consume 48.5% energy compared to the D-cache versions with a WE policy, 53.7% energy compared to the D-cache versions with a WBWA policy, and 71.9% of energy compared to the D-cache versions with a FA and a WBWA policy.
2.4.6 Summary

In summary, from our experiments on both real hardware and architectural simulators, we can derive the following important insights. First, MLP and coalesced accesses are key reasons for the shared-memory versions to outperform the D-cache versions. Second, for D-cache versions, the D-cache write policy and D-cache contention may have a strong impact on performance. Third, eliminating shared-memory usage enables higher TLP and more opportunities to allocate variables to registers, which are the key reasons for the D-cache versions to outperform the shared-memory versions.

2.5 Related Works.

To the best of our knowledge, our work is the first to study the tradeoffs between software-managed cache and hardware-managed cache on GPU architectures. The most related work is by Jia et al. [36], who observed that the L1 D-caches are not always helpful for GPGPU performance. They identified as a reason that data fetched at the cache-line granularity may consume too much bandwidth if data reuses are limited. They also propose a compiler-time approach to predict the D-cache’s performance impact and then to turn on/off the D-cache accordingly. In comparison, we focus on comparing the effectiveness between the D-cache and shared memory. For our benchmarks, the D-cache versions are developed directly from the shared memory versions such that the cached data have strong data reuse, which lead us to identify different performance tradeoffs.

Software managed caches have been widely used in different architectures. In embedded systems, they are referred to as scratchpad memory and have been well studied [61][76]. In [12], Banakar et al. performed a comparison between scratchpads and caches for embedded systems and showed that scratchpads are a promising alternative to caches. In the CellBE architecture, local stores are essentially software managed caches and there have been numerous works on how to better utilize them for performance enhancement [29][33]. Compared to these architectures, GPUs have a high emphasis on TLP and are throughput oriented rather than latency oriented. The single-instruction multiple-thread (SIMT) style execution presents quite different performance tradeoffs between software-managed and
hardware-managed caches. As shown in our experimental analysis, TLP, MLP, and coalescing have a strong performance impact while such factors are non-existent in embedded systems.

Current works on GPU memory hierarchy mainly focuses on how to improve performance. There have been approaches proposed to utilize shared memory to convert un-coalesced accesses into coalesced ones[87]. The TLP limit introduced by shared memory usage is addressed by timing-multiplexing shared memory [88]. Cache contention among warps has been observed in [66] and a cache-conscious warp scheduling algorithm is proposed to address this problem.

### 2.6 Conclusions

In this paper, we present an in-depth study to reveal interesting and somewhat unexpected tradeoffs between shared memory and L1 D-caches in GPGPU applications. We generate the code to leverage D-caches from the code utilizing shared memory so as to ensure that the same optimization techniques such as tiling are applied equally. Since TLP can significantly hide the performance impact of L1 cache misses, many subtle factors can have more profound performance impacts than cache miss rates. Through detailed case studies, we show that the key reasons for shared-memory versions to outperform D-cache versions are MLP and coalescing (i.e., bank-conflict-free accesses). The D-cache versions mainly benefit from improved TLP and using registers to store variables. On the other hand, the D-cache versions may be affected by un-coalesced cache accesses, cache-write policies and cache contention.

Among the 16 benchmarks in our study, the shared memory versions outperform the D-cache versions and consumes less energy for most of them, justifying the software complexity to manage shared memory even in the existence of D-caches. Such results hold irrespective of latency differences between the L1-D cache and shared memory, as shown by simulations and actual hardware experiments. On the other hand, for data private to each thread, storing them into registers or local memory to leverage L1 D-caches can be a better alternative to shared memory.
Chapter 3

Automatic Data Placement into GPU On-Chip Memory Resources

3.1 Introduction

Throughput-oriented architecture, such as graphic processing units (GPUs), has been widely used to accelerate many general-purpose computation workloads. Although general purpose computation on GPUs (GPGPU) achieves high throughput mainly by employing a large bundle of threads to overlap computations with long-latency memory accesses, off-chip memory bandwidth and latency remain a performance as well as energy-efficiency bottleneck. Furthermore, the current trend of GPGPU evolution scales the computational throughput much faster than off-chip memory access bandwidth. For example, Nvidia GTX480 GPUs based on the FERMI architecture [3] have an arithmetic throughput of 1.35 TFLOPSs with the memory bandwidth of 178 GB/s. In comparison, GTX680 GPUs based on the KEPLER architecture [4] have an arithmetic throughput 3.09 TFLOPS (2.3X increase over GTX480) with the memory bandwidth of 192 GB/s (7.8% increase over GTX480). To alleviate the off-chip bandwidth bottleneck and reduce memory access latency, GPUs are equipped with a multiple-level on-chip memory hierarchy including register files, L1 data caches (D-cache), shared memory, and L2 caches. As expected, how to effectively utilize such on-chip memory resources has a significant impact on application performance. However, it is non-trivial for application developers to explicitly manage these on-chip memory resources as the trade-offs among these
resources are complex and sometimes non-intuitive [57]. More importantly, as on-chip resources have been changing significantly for different generations of GPUs, an optimized kernel upon one generation becomes suboptimal on another one. Thus performance portability is a daunting challenge for application developers.

In this work, we propose compiler-driven automatic data placement as our solution. We focus on GPGPU programs that have already been reasonably optimized either manually by programmers or automatically by some compiler tools. In other words, our input programs already employ classical loop optimizations such as tiling and allocate important data, either for communication among threads or for data reuses, in shared memory. Our proposed compiler algorithm refines these programs by revising data placement across different types of GPU on-chip memory resources.

Our compiler algorithm places data into different types of on-chip memory resources using the following systematic way. First, it analyzes the usage patterns of all shared memory variables in an input kernel program and tries to promote those shared memory variables into registers if they are not used for communication among threads. Second, if the shared memory usage becomes the bottleneck for thread-level parallelism (TLP), it checks whether it is profitable to move some shared memory variables into either global or local memory so as to implicitly exploit the L1 D-cache. Third, it detects redundant accesses to both global memory and shared memory across different threads. Then, it aims to reduce such redundant accesses by compacting multiple threads into one, thus converting redundant shared/global memory accesses among threads into data sharing/reuse of registers. To find the most profitable data (re)placements, an auto-tuning process is used to select the optimal parameters in the optimization process. The first two steps of our compiler algorithm focus on replacing shared memory variables with registers or global/local memory variables. The key reason is due to the evolution trend of GPU on-chip memory resources. In early generations such as the Nvidia G80 and GT200 architecture, the ratio of the register file size to the shared memory size is 2 and 4, respectively. In comparison, in the FERMI and KEPLER architecture, the ratio becomes 2.7 and 5.3, respectively. As a result, the code optimized for G80 or FERMI tends to over-
utilize shared memory while underutilizing the register file when it runs on GT200 or KEPLER GPUs. As a result of such underutilization, it is proposed in prior works [10] to turn off significant portions of the register file to reduce static power consumption.

We evaluate our proposed automatic data replacement algorithm using a diverse set of applications from different GPGPU benchmark suites that have been manually optimized. Our results show that our compiler algorithm improves the performance by up to 4.14X and an average 1.76X on the FERMI architecture, and by up to 3.30X and an average of 1.61X on the KEPLER architecture.

The remainder of the chapter is organized as follows. Section 3.2 presents a brief background on GPU architecture with an emphasis on on-chip memory resources. Section 3.3 presents in detail our proposed automatic data placement algorithm. Section 3.4 and 3.5 discuss our experimental methodology and the experimental results. Section 3.6 addresses the related work. Section 3.7 concludes this chapter.

### 3.2 Background and Motivation

State-of-the-art GPUs employ many-core architecture, on which the cores are organized in a two-level hierarchy. Each GPU contains a cluster of streaming multiprocessors (SM) in Nvidia GPUs, or computing units in AMD GPUs. Each SM in turn consists of multiple streaming processors (SPs). To amortize the overhead of instruction fetch and decode, an array of SPs executes one scalar program in the single-instruction multiple-data (SIMD) manner. A group of threads running on such an array of SPs and sharing the same program counter (PC) is referred to as a warp of threads. Multiple warps of threads are grouped into a thread block (TB) and a number of thread blocks are organized into a thread grid.

The GPU off-chip memory space consists of texture memory, constant memory, local memory, and global memory. Texture memory and constant memory are for read-only data which can be accessed by all threads. Global memory can be read or written by all threads in a kernel. In contrast, local memory is private to each thread.
In order to reduce the latency and improve the bandwidth of off-chip memory accesses, three types of on-chip memory including shared memory, data caches, and a register file, are introduced in each SM. Texture caches and constant caches are also on-chip memory but they are used for read-only data and not our focus in this study.

Among three types of on-chip memory, the register file has the shortest access latency and highest throughput. Furthermore, the register file is larger than the L1 D-cache and shared memory as shown in Table 3.1. The register file is private to each thread, which means data in registers can only be accessed by a single thread, except for the latest Nvidia KEPLER architecture, which introduces a new instruction “__shfl” [4] to enable a thread to access the registers in other threads within the same warp. The maximum number of registers per thread is ISA-dependent and varies in different architectures. Exceedingly heavy usage of registers per thread will result in register spills into its local memory, which may be captured in L1 D-cache.

Compared to register files, shared memory has lower throughput and smaller capacity. As shown in Table 3.1, a GTX 680 GPU has a 256KB register file and 48KB shared memory. As shared memory is accessible to all threads in a TB and has low access latency, prior works have been focused on using shared memory to achieve memory coalescing, to provide data communication, and to store data for temporal reuses. L1 D-cache shares the same hardware resource as shared memory on FERMI or KEPLER architecture, in contrast to shared memory, which is explicitly managed by kernel code, L1 D-caches are hidden from developers and are implicitly managed by hardware to keep the most recently accessed data. Furthermore, while the intensive usage of shard memory or registers can limit the number of threads running on each SM, the usage of L1 D-cache does not. However, too many threads in a SM would compete with each other for the limited L1 D-cache capacity, which may result in poor performance due to cache contention [42].

GPUs evolve at a fast pace. Taking Nvidia GPUs as an example, from the first generations of unified shader G80 to the state-of-art KEPLER architecture. A comparison of them is shown in Table 3.1. Several observations can be made from the table. First, there is a higher increase
in computational throughput than off-chip memory bandwidth. For example, from the FERMI architecture to the KEPLER architecture, the computation throughput increases by up to 229% while the memory bandwidth increases by only 8.3%. As a result, we need to more carefully manage on-chip resource to effectively utilize the computational resources. Second, among GPU on-chip memory resources, the register file size and D-cache/shared memory have been changing across different generations. For example, from G80 to GT200, the register file size is doubled while the shared memory capacity remains the same. The same trend is present when comparing the FERMI architecture and the KEPLER architecture. Consequently, the code optimized for early GPU generations tend to use shared memory more heavily. This leads to a serious challenge for performance portability for such optimized code running on different GPUs.

Table 3.3. A comparison of hardware characteristics across different GPU generations.

<table>
<thead>
<tr>
<th></th>
<th>G80 (GTX 8800)</th>
<th>GT200 (GTX 280)</th>
<th>FERMI (GTX 480)</th>
<th>KEPLER (GTX 680)</th>
<th>KEPLER (K20c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic throughput (Gflops/S)</td>
<td>504</td>
<td>933</td>
<td>1345</td>
<td>3090</td>
<td>3950</td>
</tr>
<tr>
<td>Memory Bandwidth (GB/S)</td>
<td>57</td>
<td>141</td>
<td>177</td>
<td>192</td>
<td>250</td>
</tr>
<tr>
<td>Shared memory size(KB)</td>
<td>16</td>
<td>16</td>
<td>48</td>
<td>48</td>
<td>48</td>
</tr>
<tr>
<td>Register file size(KB)</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>256</td>
</tr>
</tbody>
</table>

In summary, the main challenges for application developers to manually manage the on-chip memory resources include: 1) GPUs have three types on-chip memory and, although critical to performance, it is difficult to decide the proper on-chip resource for a particular data element in an application, and 2) the resource evolution is not linear across different GPU generations, and optimal on-chip resource usage varies for different GPU generations.
3.3 Automatic Data Placement into On-chip Memory Resources

To automatically manage on-chip memory resources and achieve performance portability, in this section, we describe in detail our proposed compiler algorithm for automatic data placement. We first present our analysis of possible data placement patterns among different types of on-chip memory resources. Then, we construct our compiler algorithm using the profitable patterns.

3.3.1 Data Placement Patterns

As discussed in Section 3.2, we focus on three types of on-chip memory: register files, shared memory, and L1 D-caches. We propose to move data from one type of on-chip memory to another to achieve optimal resource utilization. As shown in Figure 3.1, there are six possible directions of moving data variables or six ways of data (re)placement. Data placement between register variables and local memory variables, i.e., direction 3 and 6, is determined by the compiler of the GPU vendors. With the Nvidia GPU compiler NVCC [6], we determine that the array variables accessed with non-constant indices, e.g., A[k] where k is a run-time variable, are allocated in local memory. Both scalars and array variables with constant indices are candidates for register allocation. Moving data from register files and D-caches (i.e., local/global variables) into shared memory, i.e., direction 4 and 5, requires significant code changes besides synchronization. Also, the current trend of GPU evolution is that the register files are much larger than shared memory and the existing compiler tools already can make use of shared memory for data reuse and communication. Therefore, we focus on placement 1, 2, and 3, and leave further investigation on placement 4 and 5 as future work.
Pattern 1: Promote variables from shared memory to registers

Shared memory can be used to exchange data among threads in a TB. Also, as a low-latency on-chip resource, many applications use shared memory as software-managed cache to hold important (private) data for each thread. There are three reasons why it may be profitable to promote a shared memory variable into registers. First, the shared memory usage may limit the number of concurrent TBs on an SM, i.e., TLP, and promoting shared memory variables into registers can alleviate the pressure on this critical resource. Second, shared memory has longer access latency and lower bandwidth than register files. Third, accessing shared memory variables is associated with instruction overhead for address computations. Therefore, higher performance may be expected when promoting shared memory variables into registers.

Figure 3.2 Data placements among three types of on-chip memory.
We show a benchmark, PathFinder, as an example, in Figure 3.2. PathFinder makes use of two shared memory arrays, ‘prev’ and ‘result’, as shown in Figure 3.2. Its TB dimension is 256x1 and its thread grid size is 19x1. As a result, the sizes of these two shared-memory arrays are small (256x4=1kB) and such shared memory usage is actually not a bottleneck for the number of concurrent TBs on each SM. For the shared-memory array ‘prev’, its accesses in the kernel code, ‘prev[tx-1]’ and ‘prev[tx+1]’ indicate that the data in this array are indeed shared among different threads. As shown in line 7 in Figure 3.2a, the ‘result’ array is accessed by each thread multiple times in a loop. As each thread only accesses the array result using its own thread id as shown in line 8 and line 9 in the figure, there is no communication using the ‘result’ array across threads. Since each thread only accesses its individual part of the array, it is safe to simply replace ‘result[tx]’ with a register. Further, as the variable is only defined and used in the same thread, we can safely remove the synchronization instruction ‘__syncthreads()’.

```c
__global__ void dynproc_kernel(…){
    __shared__ float prev[256];
    __shared__ float result[256];
    int tx=threadIdx.x;
    for (int i=0; i<iteration ; i++){
        .... shortest = minum( prev[tx-1], prev[tx],prev[tx+1]);
        result[tx] = shortest + gpuWall[index];
        __syncthreads();
        prev[tx]= result[tx];
        __syncthreads();
    }
    gpuResults[xidx]=result[tx];
} a) Baseline

__global__ void dynproc_kernel(…){
    __shared__ float prev[256];
    float result;
    int tx=threadIdx.x;
    for (int i=0; i<iteration ; i++){
        .... shortest = minum( prev[tx-1], prev[tx],prev[tx+1]);
        result = shortest + gpuWall[index]; __syncthreads();
        prev[tx]= result; __syncthreads();
    }
    gpuResults[xidx]=result;
} b) Optimized Code
```

Figure 3.3 A code example of PathFinder.
after the statement updating the variable ‘result’ (line 7). The resulting code is shown in Figure 3.2b.

In our study, we found that shared memory is used very often in many benchmarks. Therefore, there are usually multiple shared memory arrays that can be replaced with registers. In this case, we may not have enough registers to promote all the shared memory arrays, and need to decide which shared memory array should be replaced with registers to maximize the performance benefits. Our framework handles this problem by counting the references of each shared memory array, and gives higher priority to the one with larger reference counts (Section 3.3).

**Pattern 2: Promote variables from shared memory into L1 D-caches**

As discussed above, the register file cannot be used for an array variable with a dynamically determined index (e.g., A[x]) and intensive usage of registers for shared memory promotion can also limit TLP. The local memory or global memory, which implicitly utilizes the L1 D-cache to achieve the high performance, does not have such drawbacks. Therefore, promoting variables from shared memory into local memory / global memory (L1 D-cache) is a better choice when (1) replacing shared memory arrays with dynamic indices or (2) the shared memory array to be promoted has a large size (e.g., an array of structures). Furthermore, if a shared memory variable is used for communication among threads, a global memory variable can be used to replace it since global memory is visible for all threads.
Figure 3.3a, using the benchmark, Marching-Cube (MC), from CUDA SDK as an example, shows that two shared memory arrays ‘vertlist’ and ‘normlist’ are used in the kernel. Each thread only accesses part of these two arrays, and the total size of these two arrays is 9216 bytes for each TB. As a result, each SM can run 5 TBs concurrently even when the shared memory is configured to be 48KB. As we can see from the figure, the value of variable ‘edge’
in line 11 of Figure 3.3a can only be determined in the runtime, and therefore the array ‘vertlist’ cannot be allocated in the registers. We choose to promote these two arrays into local memory instead of global memory to minimize the code change since for global memory, we have to modify the CPU code to allocate a global memory array and insert it as a parameter of the kernel invocation. The resulting code is shown in Figure 3.3b. Since the code in Figure 3.3b does not use shared memory any more, each SM can run up to 16 TBs in the KEPLER GPUs and 8 TBs in the FERMI GPUs. Such improved TLP leads to higher performance for MC. In many cases, an application may intensively use shared memory to communicate among threads. Then, the global memory has to be used to replace the shared memory variables to maintain such communication so that we can both overcome the TLP bottleneck imposed by shared memory usage and keep inter-thread data communication. Note that although promoting variables from shared memory into L1 D-cache can significantly improve the TLP (or occupancy) otherwise limited by shared memory capacity, it doesn’t mean that more TLP will always lead to higher performance. In some scenarios, more concurrent TBs may increase cache and/or network contentions and adversely affect the performance. Thus, our compiler uses an auto-tuning process to determine (1) how many variables should be promoted and (2) whether they are promoted into local memory or global memory, so as to achieve optimal data placement in balancing trade-offs between TLP and network/memory pressure.

Pattern 3: Promote variables from shared memory / global memory into registers to achieve register tiling

A common side effect of single-program multiple-data (SPMD) parallelization is redundant computations and memory accesses. In GPU kernels, there often exist redundant accesses to either shared memory data or global memory data across different threads. This redundant shared/global memory reference can be promoted into register usage to further save bandwidth.

We use the benchmark SRAD as an example to illustrate this behavior. Figure 3.4a shows a code segment from the SRAD kernel code. The TB dimension of the SRAD kernel is
configured as <16,16>, i.e., 256 threads per TB. Therefore, tx (i.e., threadIdx.x) ranges from 0 to 15 for all the warps in a TB; and ty (i.e., threadIdx.y) will be 0~1 for the first warp, 2~3 for the second warp, and so on. Before computation, a tile of data will be loaded from the global memory array ‘c_cuda’ into the shared memory array ‘south_c’ as shown in line 8 of Figure 3.4a. We can see that the index variable ‘index_s’ is dependent on tx, bx (i.e., blockIdx.x) and by (i.e., blockIdx.y) but not on ty. It means that when the 8 warps of a TB actually load the same block of global memory data, there are 7 redundant global memory accesses in each TB since all the warps share the same tx, bx, and by, i.e., the same memory reference index.

All three types of on-chip memory can be potentially used to reduce the overhead of such redundant global memory accesses across warps. First, the L1 D-cache is utilized implicitly when redundant global memory accesses hit in the L1 D-cache but such data reuse cannot be assured as the data may be evicted by other data requests. Second, we can choose to let only the first warp load the data into shared memory, and other warps then access the data from shared memory. However, this way incurs overhead due to operations moving data from/into register into/from shared memory[57]. Additional control flow is also needed to ensure that the global memory data are loaded only once and a synchronization is necessary to eliminate potential data races. Third, although the register file has a large size and the lowest latency, it cannot be shared among warps. In order to take advantage of the register file, we need to first compact multiple warps/threads into a single warp/thread, and then promote shared/global memory variables into registers. This way, the register variables after thread compaction can be shared among different threads before compaction. Such thread compaction is also referred to as thread merge [87] and thread coarsening[43]. Compared to the prior works on thread merge/coarsening/fusion[87], our approach specifically leverages this optimization technique for register tiling, i.e., use register reuse to eliminate redundant shared/global memory accesses. A key question for such register tiling is how many threads to be compacted so as to maximize register reuse while restricting the register pressure on TLP. We introduce the compaction factor C_Factor in our compiler algorithm to determine the most profitable version of data placement using automatic tuning.
The optimized code after compaction is shown in Figure 3.4b. The number of original threads/warps to be compacted is defined as a run-time constant, C_Factor. First, the thread
block dimension is adjusted from <16,16> to <16,16/C_Factor>. Second, the global memory read accesses on line 7 of Figure 3.4a are replaced with a single global memory access on line 6 of Figure 3.4b, which loads the data from global memory to the register variable ‘tmp_1’. Third, since multiple threads/warps of Figure 3.4a are compacted into a single thread/warp in Figure 3.4b, we can reuse the register ‘tmp_1’ as shown in line 12. Similarly, the memory access of ‘c_cuda’ under the conditional statement (line 8 of Figure 3.4a) can be processed in the same way by introducing another register ‘tmp_2’ as shown in Figure 3.4b. The if statement in line 8 of Figure 3.4a sometimes may also need to be replicated to guard this ‘c_cuda’ access to avoid potential out-of-bound accesses.

3.3.2 Compiler Algorithms and Implementation

Although the data placement patterns discussed in Section 3.3.1 can be used to guide programmers to manually optimize their GPU programs, it will quickly become unmanageable if a non-trivial number of data variables are to be analyzed. In this section, we present our source-to-source compiler framework which implements these three data placement patterns using an automatic compiler optimization algorithm. The goal of the compiler algorithm is to generate the code which utilizes on-chip resource efficiently without effort from application developers. The key feature is that the compiler framework can intelligently re-assign the memory types of variables of a GPU program to maximize the benefit of on-chip resources. Our compiler algorithm has two passes: one for data placement pattern 1 and pattern 2 and the other one for data placement pattern 3.

Either compiler pass has three stages: the identifying stage, the processing stage, and the auto-tuning stage, as detailed in Figures 3.5 and 3.6. The identifying stage will scan all the memory variables, and generate a list of candidate variables which can be promoted by collecting the architecture features and analyzing the memory accesses of the target kernel. The processing stage implements the data placement patterns by revising the data types and their access indices of these candidate variables. The auto-tuning stage constructs the search spaces, decides which variables to be processed and selects the optimal code versions.

Compiler pass 1
The algorithm of the compiler pass for promoting shared memory variables to register files/local memory/global memory is shown in Figure 3.5. The identifying stage (line 5~15) collects all shared memory variables through their ‘__shared__’ keyword. For shared memory variables, we mark an access as a combination of the array name and the access index. The compiler checks access indices to determine (a) whether an access is across different threads.
or private to a single thread, and (b) whether an index has to be determined at the runtime. Meanwhile, the reference count of the variable is also recorded. If an access is inside a loop, we weight this access number by timing a loop count in line 10. In some cases, the loop count in a one-level loop or multiple loop counts in nested loops may associate with a run-time value, leading to some unknown reference counts. In such cases, we resort to either profiling or simple heuristics (Section 3.2.4). The output of the identifying stage is arrays, a list of candidate variables.

For all the candidate variables in arrays, the processing stage (line 18 ~24) applies data placement patterns by first selecting the shared memory variable with the largest reference counts. Then, if a shared memory variable is not shared across threads and is not accessed with run-time determined indices, it is promoted to the register file. Otherwise, it is replaced with a global memory variable if is used for inter-thread communication in line 23~24; or it will be replaced with a local memory variable if it is accessed through indices at line 21~22. Each replacement will result in substituting both the variable definition and reference indices throughout the kernel code from original one to the promoted type.

**Compiler pass 2**

The second compiler pass implements the third data placement pattern, i.e. promoting redundant shared/global memory accesses into register accesses, as shown in Figure 3.6. In the identifying stage (line 5~13), the compiler analyzes each shared or global memory array. It checks whether an array index is independent upon the thread id in either the X or Y dimension. If it is independent upon both dimensions, it sets the flag is_redundant_2d. Otherwise, if it is independent upon one direction, it sets the flag is_redundant_1d. During each index check, the compiler also inserts the expressions associated with the index into the exprs list, which will be used in the processing stage. After the identification stage, it outputs exprs, the list of candidate expressions that exhibit data access redundancy, and the corresponding flags that indicates the type of redundancy type, i.e., one-dimension or two-dimension.
In the processing stage (line 16~27), the compiler first adjusts the thread block dimension for each different compact factor (C_Factor) in line 16. Then, it constructs an unroll-able loop to perform the workloads of the threads to be compacted. The loop body contains all the expressions including the associated computational operations and memory accesses in the

Figure 3.7. The compiler algorithm to promote shared or global memory to registers to be shared among threads.
thread index dependence chain, as shown in line 18~19 and line 23~25. The exception is the expressions in the expr lists collected in the identification stage. These global/shared memory accesses in the expr list are performed only once by loading data into a destination register as shown in line 20~21 and 26~27 and the destination register will be reused in the newly constructed loop, as illustrated in Figure 3.4. This C_Factor is a tunable parameter to indicate how many threads/warps will be compacted into one. The C_Factor can be a scalar or a two-dimension vector depending on the redundancy type generated from the identifying stage.

**Auto-tuning**

In the auto-tuning stage, the compiler first generates a search space based on all the tunable parameters, then measures the execution time, compares them, and finally selects the best performing version. Totally, three search spaces will be generated associated with each data placement policy. The first search space is to decide how many and which shared memory variables to be promoted into the register file; the second search space is which shared memory variables to be promoted into global/local memory; the third search space is to determine the compaction factor.

To manage the cost of auto-tuning, we prune the first search space by promoting shared memory variables incrementally, starting from the one with the highest reference counts, assuming that this one will benefit most when being promoted into registers. If one version has lower performance than the previous one, it means that promoting one more shared memory variable may lead to too much register usage and hurt the performance. Therefore, it stops further promotions. For the second search space, we prune it by using a greedy strategy to promote shared memory variables that occupy the largest space, so that it will release the resource pressure on TLP in a fast and incremental way. For the compaction factor, we observe that, the thread block in GPU computing workloads is typically a multiple of 32. Therefore, we constrain the compaction factor as a number of 2’s power. Compared to the previous sophisticated methods for pruning the search space such as generic algorithm [17] and machine learning techniques [71], our heuristics are simple and practical on GPU kernels. In Section
3.5, our results also show that for our workloads, our iterative space pruning approach is effective in reducing search space and finding the optimal/near optimal version.

The auto-tuning part for compiler pass 1 is listed as line 27 to line 32 in Figure 3.5. During auto-tuning, if a newly generated kernel has worse performance than the previous version, the compiler will consider further optimization is not helpful and the previous version is chosen as the best one as shown in line 32. In Figure 3.6, the code lines from 29 to 35 show the auto-tuning part for compiler pass 2. The compiler evaluates the new kernel generated by previous steps in line 30. If the new kernel has better performance, then the compiler increases the C_Factor and continues with more aggressive thread compaction. Otherwise, the compiler stops at line 35.

**Preprocessor**

In our implementation, our compiler framework takes a pre-processing step on the program and regulates expression representation for successive analysis and optimization. First, the index for an array access is interpreted as an affine function of the thread index. The scaling factor in the affine function may involve a subset of the kernel launch parameters, macro/constant values, run-time parameters, and loop iterators if the memory expression is inside a loop. Second, an array access may reside inside a condition or loop statement. The reference count of such an array depends on the loop bound and the condition. If the loop bound and the condition can only be determined at run-time, we choose to either let the user to provide such information through profiling or use the following simple heuristics. We assume that for a nested loop in a kernel, each level has a loop count of 4 and the condition is true half of the times. The reason for such a default loop count is that our observation from the benchmarks shows that when a nested loop is parallelized into GPU threads, the levels with large loops counts are used to generate thread grids and the thread body typically contains loops with smaller counts. Lastly, the preprocessor collects the data structure declaration and annotate array accesses with the data type. For the vector data type such as int2, float4, the memory access index is processed the same as the scalar data type. For the struct type, the array index and the addresses of its elements are identified separately.
3.4 Experimental Methodology

We implemented our compiler algorithms using Cetus [55], a source-to-source compiler infrastructure for C programs. The CUDA syntax support is ported from MCUDA [72].

<table>
<thead>
<tr>
<th>Parameter</th>
<th>GTX480</th>
<th>GTX680</th>
<th>K20c</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Shared memory size, L1 D-cache size&gt;</td>
<td>&lt;16kB, 48kB&gt;, &lt;48kB, 16kB&gt;</td>
<td>&lt;16kB, 48kB&gt;, &lt;32kB, 32kB&gt;, &lt;48kB, 16kB&gt;</td>
<td>&lt;16kB, 48kB&gt;, &lt;32kB, 32kB&gt;, &lt;48kB, 16kB&gt;</td>
</tr>
<tr>
<td>Register file size</td>
<td>128kB</td>
<td>256kB</td>
<td>256kB</td>
</tr>
<tr>
<td>Max number of threads per SM</td>
<td>512</td>
<td>1024</td>
<td>1536</td>
</tr>
<tr>
<td>Max number of registers per thread</td>
<td>64</td>
<td>64</td>
<td>256</td>
</tr>
<tr>
<td>Compaction Factor</td>
<td>2,4,8,16</td>
<td>2,4,8,16</td>
<td>2,4,8,16</td>
</tr>
</tbody>
</table>

To evaluate our proposed compiler optimizations, we perform our experiments on Nvidia GTX480 (FERMI) GPUs, GTX680 (KEPLER) GPUs, and Telsa K20C GPUs. The parameters are presented in Table 3.2.
Most of the benchmark kernels used in our experiments are from Rodinia [17] and CUDA SDK [5] since they have already been manually optimized. Among them, HotSpot, Back Propagation, SRAD, Pathfinder, B+tree, LU Decomposition are from the latest Rodinia suite. Matrix Multiplication and MarchingCubes are from CUDA SDK. NQU is from GPGPUsim benchmark suite [11]. As Back Propagation, SRAD and B+tree, contain two GPU kernels, we use BackPropagation1, BackPropagation2, SRAD1, SRAD2, B+tree1, B+tree2 to differentiate them. In Table 3.3, from left to right, we show the benchmark name, the input, as well as the resource usage including the number of registers per thread and the size of shared memory (bytes) per SM on GTX 480, GTX 680, and Telsa K20c, respectively. We use the default input released with the code. For each benchmark, the shared memory usage is the same for different GPUs because it is determined by programmer’s explicit definition. The register usage is statically allocated and the maximum available registers per thread vary on different GPUs.

3.5 Experimental Results

In our first experiment, we measure the execution time of both the original kernel and the optimized kernel generated from our compiler algorithm on GTX480, GTX680 and Telsa K20c
separately. On each GPU, we tried all different shared memory/L1 D-cache configurations and selected the one with the best performance for the original kernels. Also, for each optimized kernel, the compiler will generate the best data placement to accommodate the specific architecture so as to achieve optimization portability. Each benchmark has been run one-hundred times to obtain the stable execution times. Figure 3.7 shows performance comparisons between original kernels and our optimized ones across different GPUs.

![Figure 3.7. Performance speedups achieved by automatic data placement for all benchmarks on three different GPUs.](image)

From Figure 3.7, we can see that across all the benchmarks, the optimized kernels exhibit significantly higher performance than the original ones on all the three GPUs. The benchmark SR1 achieves the highest speedup (4.14X) over the original kernel on GTX480. This is because most global memory accesses in this kernel are redundant, not only among different warps but also among threads in a warp. By promoting those redundant global memory accesses into register accesses, the access time to all input data sets is highly reduced. Similar global or shared memory redundancy also exists in HS, BP1, BP2, MM, BT1, BT2, and LUD. For HS, PF and NQU. Many shared memory variables in these kernels have no data exchanges among threads, thus these shared memory variables are candidates for register promotion. In NQU, there are five shared memory variables and four of them are promoted, leading to a high performance speed-up of 3.3x on GTX680. For PF, _syncthreads() can be safely removed. However, even though it is not removed, the optimized code (e.g., on GTX480) can still
achieve 7% performance improvement. For MC, shared memory variables holding two on-chip working sets can be promoted into local memory arrays so as to remove the resources limitation on the number of concurrent TBs, thereby achieving higher performance. Overall, using the geometric mean as an average, the kernels optimized on GTX480 can achieve up to 4.14x speedup and an average of 1.76x speedup compared to the original benchmarks, up to 3.30x speedup and an average of 1.61x speedup on GTX680, and up to 2.44x speedup and an average of 1.48x speedup on K20c.

![Diagram](image)

**Figure 3.8. Auto-tuning of our automatic data-placement for all benchmarks on GTX680 (Performance normalized to original kernel).**

In our second experiment, we first breakdown the effectiveness of each placement pattern. Figure 3.8a and 3.8b shows the benchmarks that can be applicable to compiler pass 1 and pass 2. Among them, only HS benefits from both pattern 1 and pattern 3 (the total improvement of 64.2%: breakdown into 4.8% from pattern 1 and 59.4% from pattern 3), while other
benchmarks only benefit from one in three patterns: MC benefits from pattern 2; PF, NQU benefit from pattern 1, and others benefit from pattern 3.

![Figure 3.9. The optimal parameter, the number of shared memory array to be promoted and the C-Factor, determined for different GPUs.](image)

We further evaluate the effectiveness of our auto-tuning process for each benchmark. As shown in Figure 3.8a, the benchmarks NQU, PF, HS and MC benefit from promoting shared memory arrays into register/local/global memory. The search space is how many shared memory variables can be promoted into registers or L1 D-cache using our compiler pass 1 in Section 3.2.1. For all the cases, promoting more shared memory variables into registers or L1 D-cache will lead to higher performance. For the benchmark kernels benefiting from reduced redundant shared/global memory accesses, Figure 3.8b shows the impact of the search parameter C_Factor in our compiler pass 2 in Section 3.2.2. From Figure 3.8b, we can see that the best C_Factor varies across different benchmarks. For SR1, the best performing version is achieved when C_Factor is 16. However for BP1, the best performing one is obtained when C_Factor is 2, and further increasing C_Factor to 4 degrades the performance as it reduces the number of active warps in a thread block. Such reduced TLP subsequently degrades the latency hidden ability for off-chip memory accesses, offsetting the profit from reducing redundant accesses. Therefore, auto-tuning is stopped when such a performance drop is observed. We can see that if C_Factor is increased to 8 for BP1, the performance will degrade even more.
This validates the effectiveness of our auto-tuning policy, which searches C_Factor in an incremental manner. The same scenario has also been observed in the compiler pass 1 from Figure 3.8a when searching for the appropriate shared memory variables to be promoted in MC.

Third, in Figure 3.9, we present the optimal parameters determined by our auto-tuning process on the different GPUs. For PF, NQU and MC, the y-axis means how many variables should be promoted while for others, the y-axis denotes the optimal C_Factor values on different GPUs. Our compiler can intelligently generate the optimized kernel for specific architecture to achieve optimization portability. We can see that the different architecture features of these GPUs lead to different optimal parameters. For example, NQU achieves best performance when its four shared memory variables are promoted on GTX680, while on K20C, the best performance is achieved when three shared memory variables are promoted.

<table>
<thead>
<tr>
<th>original search space</th>
<th>Pruned search space</th>
<th>Auto-tuning time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HS 48</td>
<td>8</td>
<td>42.873</td>
</tr>
<tr>
<td>BP1 16</td>
<td>3</td>
<td>11.361</td>
</tr>
<tr>
<td>BP2 16</td>
<td>4</td>
<td>15.755</td>
</tr>
<tr>
<td>SR1 16</td>
<td>5</td>
<td>24.133</td>
</tr>
<tr>
<td>SR2 16</td>
<td>5</td>
<td>21.941</td>
</tr>
<tr>
<td>MM 32</td>
<td>5</td>
<td>210.876</td>
</tr>
<tr>
<td>PF 1</td>
<td>1</td>
<td>8.88</td>
</tr>
<tr>
<td>NQU 45</td>
<td>12</td>
<td>48.124</td>
</tr>
<tr>
<td>MC 9</td>
<td>6</td>
<td>23.986</td>
</tr>
<tr>
<td>BT1 3</td>
<td>3</td>
<td>12.183</td>
</tr>
<tr>
<td>BT2 3</td>
<td>3</td>
<td>14.343</td>
</tr>
<tr>
<td>LUD 16</td>
<td>4</td>
<td>129.531</td>
</tr>
</tbody>
</table>

Fourth, our auto-tuning process has a low overhead on searching the optimal parameters. We report the cost of the auto-tuning function in Table 3.4. From the left to right, we report
the search space, i.e., the number of all possible values to be tried, in the original search if there is no pruning strategy in searching, the search space after applying our pruning strategies in our compiler passes, and the total execution time of our auto-tuning process for generating the optimal kernel for each benchmark. We can see that the search space is reduced significantly by our pruning strategy. We also validated that the optimized one from our pruned space can achieve the same performance as the one from the original search space.

![Graph showing performance speedup of optimized kernels in Marching Cubes with different input sizes.](image)

**Figure 3.10. Performance speedup of optimized kernels in Marching Cubes with different input sizes.**

Finally, besides the kernel code itself, we also consider how the problem input of a workload affects our proposed optimization process. For our first compiler pass, the shared memory array sizes are fixed with constants or macro variables which are independent of input sizes. The reason is that the benchmark code has been already optimized to process the inputs as tiled working sets. For the second pass, the input size will impact on the number of thread blocks in a grid and each thread block usually has a pre-defined size to work on a tile of input elements. Thus, the variation of the input size will not affect the steps of our compiler analysis and optimizations. Provided that the performance is in general correlated with the input size, our performance improvement will higher when the problem size is larger. Because the larger inputs will often lead to more frequent on-chip memory resource accesses to process them and our optimized kernel will in turn benefit more from the optimized access patterns. Figure 3.10
shows the effect of increased input size, from 8K Voxels to 512K Voxels, on Marching Cubes. As the input problem size increases, the performance improvement of our optimized kernel from compiler also increases from 1.179x to 1.446x.

3.6 Related works

In recent years, GPUs have been widely used for general-purpose computation due to their high computational throughput. However, achieving high performance on GPUs is not easy, and one of reasons is the intricate on-chip memory resources. Among on-chip resources, shared memory is controlled by users, and many highly optimized applications or algorithms on GPUs utilize shared memory carefully [49][80][86] so as to enjoy the low-access access latency and high bandwidth. Besides them, [49] analyses the upper performance bound of SGEMM on GPUs and optimizes the kernel through register blocking by reusing data in registers as much as possible for maximal throughput. However, none of these works considers the overhead of intensive usage of shared memory and the impacts of varying on-chip resources across different GPU generations.

To relieve the burden of optimizing GPU programs from the programmers, many auto-tuning frameworks [16][89] have been developed to automatically optimize the GPU programs to achieve high performance. For example, a polyhedral model is used in [16] for optimizing global memory accesses. In [88], the shared memory is time multiplexed to reduce the pressure on limited shared memory capacity. In [89], language and compiler support are proposed to leverage nested parallelism inside the GPU programs. However, most of these works focuses on optimizing memory accesses and managing thread-level parallelism using compiler techniques. Management of different types of on-chip memory, especially the varying on-chip memory across different GPU generations, has not been the focus. To the best of our knowledge, our work is the first compiler algorithm to automatically optimize data placement across different on-chip memory resources in a systematic way.

We also observed that vendor’s compiler may promote the variables in shared memory to register file. The way to avoid such an optimization is to use the ‘__volatile__’ keyword when declaring a shared memory array. However, as we verified from the assembly codes, we found
that the vendor’s compiler does not apply such optimizations on the benchmarks used in our work.

Current studies on on-chip memory resources mainly focus on identifying resources limitation and boosting the performance by improving architecture design [31][32]or compiler support [36]. On-chip data cache may lead to cache contention and [36] proposes a compiler algorithm to automatically turn on/off the D-cache by predicting how cache will affect the performance. The register usage pattern is studied in [31] and the register file accesses are reduced by proposing a register file cache. However, these works target on optimizing one specific resource to conquer their limitations instead of balancing on-chip resources.

The trade-offs between software-managed shared memory and hardware-managed D-cache on GPUs have been studied in [57]. Gebhart et al. [32] made the observation that different applications have different needs for various memory resources. They proposed unified local memory that can dynamically change the partition among registers, cache, and shared memory according to each application’s needs. Hayes and Zhang [34] proposed unified on-chip memory allocation which uses shared memory to offload register pressure. In comparison, our work focuses on re-assigning data across all on-chip memory resources.

3.7 conclusion

Judicious utilization of the on-chip memory resources has a significant impact on application performance. However, how to manage these intricate on-chip memory resources is non-trivial for application developers. More importantly, the varying on-chip resource across different GPU generations makes performance portability a daunting challenge. In this paper, we propose compiler-driven automatic data placement as our solution. We focus on GPGPU programs that have already been reasonably optimized either manually by programmers or automatically by existing compiler tools. Our proposed compiler algorithms refine these programs by altering data placement among different on-chip resources to achieve both performance enhancement and performance portability. In particular, we leverage three data placement patterns. First, we explore shared memory variables to promote them into registers. Second, we explore the opportunities to utilize the L1 D-cache by promoting variables from
shared memory into global/local memory if shared memory is a resource bottleneck. Third, we eliminate redundant shared/global memory accesses across different threads. To achieve performance portability, our compiler performs auto-tuning on different GPUs to achieve optimal performance. Among the benchmarks in our study, our proposed compiler algorithms significantly improve the performance by up to 4.14x and 1.76x on average on Nvidia GTX480 (i.e., FERMI) GPUs, and by up to 3.30x and 1.61x on average on GTX680 (i.e. KEPLER) GPUs, and up to 2.44x speedup and an average of 1.48x speedup on K20c GPUs. Our compiler-optimized kernel can also save up to 74.3% energy and save an average of 40.3% energy overall measured on GTX680 GPUs.
Chapter 4

Locality-Driven Dynamic GPU Cache Bypassing

4.1 Introduction

Massively parallel, throughput-oriented processors such as graphics processing units (GPUs) leverage high thread-level parallelism to overlap long latency memory accesses with computation. On-chip caches, especially L1 data caches (L1 D-caches) which were designed to improve the performance of irregular workloads without programming scratchpads, remain critical to provide high-bandwidth low-latency data accesses. However, the limited L1 D-cache capacity becomes a performance bottleneck as the working set of the massively threaded applications often exceeds the L1 D-cache capacity, causing severe thrashing[66]. More importantly, large number of the incoming memory requests with no or low reuse may evict cache lines with high reuse, resulting in cache pollution. In this case, even advanced cache replacement policies (e.g. RRIP [38] and SHiP [82]) are ineffective to address such contention problems on GPUs[66]. Furthermore, the massive multithreading on GPUs can cause the unpredictability of cache locality and various resource congestion (e.g. MSHR allocation failures) [39].
To alleviate contention and avoid early eviction, cache bypassing schemes have been proposed for CPUs [30][83] and GPUs[39][84]. The CPU-based approaches are usually designed for last level caches (LLCs), where data locality is already filtered by previous level(s) of caches. But the poor locality of GPU workloads and resource congestion impose difficulty for them to make robust predictions and they often increase L2 and DRAM level traffic(Section 4.6.1(a)). GPU-based bypassing schemes are generally conditional/reactive bypassing (e.g., bypass upon un- available resources [39]or static coarse-grained bypassing across all threads [84]) which can incorrectly bypass accesses with good reuse and cause memory pipeline stalls (Section 6.1(a)). None of the above approaches is a preventive scheme considering the uniqueness of the data locality of GPU access streams, which often contain a non-trivial number of requests with no/low reuse and/or distant re-reference intervals caused by frequent bursts of references (Section 3.2). Moreover, a fully-adaptive bypassing scheme is required to maintain the efficiency of workloads with good caching behavior, which are often neglected by previous approaches (Section 6.1(b)).

In this work, we propose a locality-driven dynamic bypassing design that automatically filters the access stream on cache insertions based on reuse frequency of accesses, so that only the data with high reuse and short reuse distances are stored in the L1 D-cache. For area and energy efficiency, we propose to decouple the tag and data stores of the existing L1 D-cache and integrate the locality filtering capability into the tag store through simple and cost-effective hardware extensions. Our design uses separate replacement policies to manage the decoupled tag and data store, such that the reuse information of the program and temporal locality of the data lines can be well preserved. Overall, this paper makes the following contributions:

1) Through a detailed analysis on the reuse characteristics of GPU workloads, we identify the key inefficiency of the conventional thrashing and stall-prone GPU cache design: irregular cache-unfriendly memory accesses resulting in contention at various cache levels.

2) We propose a locality-driven dynamic bypassing solution that is cost-effective and requires no profiling or runtime prediction. We demonstrate that our design can significantly
improve the performance and energy efficiency of irregular cache-unfriendly workloads, while maintaining the efficiency for regular workloads with favorable caching behavior.

3) Our design achieves significant performance improvements over the baseline caches and outperforms the state-of-the-art CPU (PDP-best[28]) and GPU (MRPB [39]) cache bypassing schemes, by dramatically reducing various types of cache contention without generating extra L2 and DRAM traffic.

The remainder of the chapter is organized as follows: Section 4.2 discusses the background, the GPU memory hierarchy in particular. Section 4.3 dissects the effectiveness of L1 D-caches for various workloads. Section 4.4 details our new design. Section 4.5 presents the experimental methodology. The evaluation of our design is shown in Section 4.6. Section 4.7 discusses the related work and Section 8 concludes.

4.2 Background

4.2.1 Baseline Architecture

This work proposes microarchitectural improvements to a massively parallel processor such as GPU architectures. Such processor consists of multiple SIMD cores, also known as streaming multiprocessors1 (SMs) in NVIDIA GPUs or Computing Units in AMD GPUs. As shown in Figure 4.1, each SM follows the single instruction-multiple threads (SIMT) execution model by fetching and decoding each instruction for a group of threads called a warp. All threads in a warp execute in lock-step in the SIMT backend. In the issue stage, a warp scheduler will select one of the ready warps to issue into the computing/memory pipeline stage. GPUs provide multiple types of memory units to improve the memory bandwidth such as L1 D-caches and shared memory (or scratchpads). Global memory and scratchpad accesses are served through L1 D-cache and shared memory respectively. Both L1 D-cache and shared memory utilize the same hardware structure and the capacity can be configured through a runtime API. While shared memory can be explicitly managed by programmers, L1 D-cache is implicitly controlled by hardware to exploit data locality. As GPU workloads with irregular memory access patterns are becoming prevalent, effective utilization of explicitly managed
memory becomes difficult. This in turn increases the importance of efficient L1 D-cache designs. All the SMs are connected by an interconnected network to the partitioned memory module, each with its own L2 data cache and DRAM partition. To save bandwidth [67], L1 is typically write through, with either write-allocate [2] or write-no-allocate [3][4], while L2 is write back with write-allocate. Victim caches and hardware prefetching are traditionally not enabled in GPUs [14].

![Memory request handling in the baseline architecture.](image)

**4.2.2 Baseline Memory Request Handling**

When a memory instruction is dispatched into the memory pipeline, the load store (LD ST) units will identify the memory access type and dispatch it into different memory units (shared memory, L1 D-cache, etc.). The requests for global and local memory data from threads in the
same warp will go through the coalescing unit to generate as few L1 D-cache line-sized requests as possible. Then for these requests, there are two possible paths depending on whether an access is cacheable, as shown in Figure 4.1. For cacheable accesses, the first path, which sends the memory requests into L1 D-cache and is labeled as `L1 D-path', is used. On a cache hit, a request will be served by sending data to the register file immediately. On a cache miss, the miss handling logic will first check the miss status holding register (MSHR) to see if the same request is currently pending from prior ones. If so, this request will be merged into the same entry and no new data request needs to be issued. Otherwise, a new MSHR entry and cache line will be reserved for this data request. A cache status handler may fail on resource unavailability events such as when there are no free MSHR entries, all cache blocks in that set have been reserved but still haven’t been filled, the miss queue is full, etc. If any of these events occurs, the memory pipeline will stall and this request will retry every cycle until needed resources are freed. Considering the small number of cache lines and MSHR entries, these resources can be quickly occupied if all memory requests are diverted into L1 D-Path.

The second path is for un-cacheable accesses, such as global memory accesses in NVIDIA’s Kepler architecture [4](not cached in Kepler’s L1). It diverts the memory requests to bypass the L1 D-cache (labeled as ‘Bypass Path’ in Figure 4.1) and directly sends requests through an interconnect into the next level memory hierarchy. On the response fill from the return queue, there are two paths to fill the data correspondingly. If the original memory request follows the L1 D-Path, then the response data will be filled into the reserved cache line in L1 D-cache and the corresponding MSHR entry is marked as filled (‘Fill Path 1’). Otherwise, the data will directly write back into the register file (‘Fill Path 2’). Compared to the Bypass Path, the L1 D-Path lowers access latencies if requested data already resides in the cache. However, memory accesses following this path may incur high stall cycles for resource contention due to massive parallelism, and also thrash the cache by evicting out the data lines that may be reused shortly. Ideally, an efficient locality monitoring mechanism should exist to divert memory requests with high data reuse into the L1 D-cache, while other requests that have no or low data reuse will be directed to the Bypass Path instead of contending on the cache resources.
4.3 GPU CACHE INEFFICIENCY AND WORKLOAD ANALYSIS

Table 4.1. Application categorization based on cache bypassing impact. CNF: Cache Unfriendly, CI: Cache Insensitive, CF: Cache Friendly. IPC is normalized to the case of all taking L1 D-Path (N-IPC). The selected applications include Particular Filter (PTF), Srad2 (SD2), Needleman-Wunsch (NW), Single-Source Shortest Path (SSSP), LU Decomposition (LUD), Hotspot (HS), Barnes-Hut (BH), CFD Solver (CFD), Leukocyte (LFK), Gaussian Elimination (GS), Fourier Transformation (FFT), Myocyte (MYC), PathFinder (PF), Srad1 (SD1), Heartwall (HT), Matrix Multiplication (MM), B+Tree (BT), and Back Propagation (BP).

<table>
<thead>
<tr>
<th>Applications</th>
<th>N-IPC</th>
<th>Instructions</th>
<th>Type</th>
<th>Suite</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTF</td>
<td>1.36</td>
<td>7022693</td>
<td>CNF</td>
<td>9</td>
</tr>
<tr>
<td>SD2</td>
<td>1.36</td>
<td>740966400</td>
<td>CNF</td>
<td>9</td>
</tr>
<tr>
<td>NW</td>
<td>1.24</td>
<td>1617408</td>
<td>CNF</td>
<td>9</td>
</tr>
<tr>
<td>SSSP</td>
<td>1.21</td>
<td>385881828</td>
<td>CNF</td>
<td>8</td>
</tr>
<tr>
<td>LUD</td>
<td>1.17</td>
<td>5875200</td>
<td>CNF</td>
<td>9</td>
</tr>
<tr>
<td>HS</td>
<td>1.05</td>
<td>109686484</td>
<td>CNF</td>
<td>9</td>
</tr>
<tr>
<td>BH</td>
<td>1.04</td>
<td>7909748</td>
<td>CNF</td>
<td>8</td>
</tr>
<tr>
<td>CFD</td>
<td>1.00</td>
<td>9308160</td>
<td>CI</td>
<td>9</td>
</tr>
<tr>
<td>LFK</td>
<td>1.00</td>
<td>7516393326</td>
<td>CI</td>
<td>9</td>
</tr>
<tr>
<td>GS</td>
<td>1.00</td>
<td>12648</td>
<td>CI</td>
<td>9</td>
</tr>
<tr>
<td>FFT</td>
<td>1.00</td>
<td>104005632</td>
<td>CI</td>
<td>33</td>
</tr>
<tr>
<td>MYC</td>
<td>1.00</td>
<td>26986736</td>
<td>CI</td>
<td>9</td>
</tr>
<tr>
<td>PF</td>
<td>1.00</td>
<td>649231040</td>
<td>CI</td>
<td>9</td>
</tr>
<tr>
<td>SD1</td>
<td>0.97</td>
<td>8281816</td>
<td>CF</td>
<td>9</td>
</tr>
<tr>
<td>HT</td>
<td>0.55</td>
<td>8811310884</td>
<td>CF</td>
<td>9</td>
</tr>
<tr>
<td>MM</td>
<td>0.54</td>
<td>58851328</td>
<td>CF</td>
<td>21</td>
</tr>
<tr>
<td>BT</td>
<td>0.41</td>
<td>542310058</td>
<td>CF</td>
<td>9</td>
</tr>
<tr>
<td>BP</td>
<td>0.23</td>
<td>190054784</td>
<td>CF</td>
<td>9</td>
</tr>
</tbody>
</table>

To evaluate the efficacy of the GPU caches, we first characterize a wide range of GPU applications (shown in Table 4.1) covering various cache sensitivity and memory access patterns (e.g., applications with highly irregular access patterns such as BH and SSSP). These applications are selected from multiple widely used benchmark suites (including GPGPUsim benchmarks[11], CUDA SDK [5], AMD SDK [1], Rodinia [17], and Lonestar suites [15], etc) and represent production GPU codes which are optimized and hand-tuned using explicitly
managed scratchpad memory extensively. We quantify how much performance improvement, which is measured using normalized instruction per cycle (N-IPC), each application can gain if all memory requests have taken the Bypass Path compared to all taking the L1 D-Path in Figure 4.1. This IPC improvement is shown in Table 4.1, where the eighteen applications are sorted in descending order. Applications whose IPC improvement is greater than 1 are classified as Cache Unfriendly (CNF), indicating the current L1 D-cache management has detrimental impact on their performance. Applications whose IPC values are not impacted are classified as Cache Insensitive (CI), as whether having a L1 D-cache or not has negligible effects on their performance. The remaining applications whose IPC improvement is less than 1 are Cache Friendly (CF): workloads that perform better if all accesses go through L1 D-cache. Section 4.5 provides a detailed description of these applications and the baseline architecture the results from this section were collected from.

4.3.1 GPU Cache Inefficiency

The memory access stream of GPU workloads has two characteristics: a mix of data requests with different reuse frequencies and different reuse distances. For instance, in the Fermi architecture, incoming accesses will enter into the L1 D-cache without being checked on whether they have future reuse. All data accesses contend with each other for limited cache resources, resulting in memory pipeline stalls. To the other extreme, similar to the bypassing approach used in Table 4.1, Kepler will bypass all global memory accesses from L1, only handling register spills to the local memory.
In general, there are three types of contention at the L1 D-cache level: (a) Inter-warp contention: capacity misses. Current GPU architectures (e.g. Fermi and Kepler) commonly have 16~48 KB capacity of L1 D-cache[3][4], but the memory footprint of the applications is typically one to two orders of magnitude larger, which causes severe thrashing. Data blocks get evicted out frequently before any reuse happens, especially when the reuse distance is long. More importantly, memory requests with no reuse may evict cache lines that have high reuse, resulting in cache pollution. (b) Intra-warp contention: conflict misses. This is caused by the concurrent threads within the same warp (32 threads) accessing to the same cache set (current GPU L1 has much lower set-associativity than 32). (c) Other resource congestion. They include resource allocation failures from limited MSHRs and miss queue entries. These resources can be quickly occupied and cause stalls if data requests come into L1 without being filtered based on their reuse patterns.

To reduce the contention described above, we apply the most direct optimization, which is increasing the capacity and associativity of the L1 and L2 caches, and observe how it affects overall performance. We intentionally increase the cache associativity and capacity to a large value, which is expensive and impractical to implement in real GPU cache designs due to increased access latency, area, and power consumption. Figure 4.2 shows several observations.
for the CNF applications, which encountered the most cache contention: (1) Compared to the L1 D-cache, CNF’s performance is insensitive to the capacity and associativity increase of L2, which indicates that L2 is not the major performance bottleneck. (2) CNF’s performance can be improved by increasing L1 D-cache’s size and associativity (to full associativity), but it is mainly bounded by cache capacity. (3) Even with these impractical cache configurations, the performance improvements for some CNF applications are still not significant, including NW, LUD, HS, and BH. These findings motivate us to develop a cost-effective cache bypassing mechanism for GPUs to maximally reduce the contention by only letting the most useful data into the L1 D-cache.

The state-of-the-art bypass policy, protection distance prediction (PDP)[28], has been proposed for CPU LLCs. This technique applies a protection distance (PD) counter to each cache line and uses it to time how many accesses are left for this line to be protected. If PD = 0, this line is marked as unprotected and can be replaced by the incoming accesses. If there are no unprotected lines left, cache is bypassed. However, applying PDP-based approaches (static or dynamic) to a GPU L1 D-cache can encounter several problems[19]. First, due to warp interleaving and resource congestion in GPUs, as a hit-rate based scheme designed for single thread processors, PDP may not improve the performance of CNF applications because the hit rate does not directly correlate to performance in GPUs (see Section 4.6.1(c)). Second, by augmenting a cache line with a protecting distance, PDP is susceptible to bypass the blocks with frequent data reuse and short reuse distances, but keeps the staled blocks with very long reuse distances for a protection interval (see Section 4.6.1(d)), causing cache pollution. Third, many of the CF applications shown in Table 4.1 have been optimized for cache performance. Therefore, any request bypassing caused by inaccurate PDP prediction can result in a significant performance degradation (see Section 4.6.1(b)). Finally, it is difficult for PDP to predict applications with irregular memory accesses, such as BH and SSSP (see Section 4.5 and 4.6.1(a)).
4.3.2 Impact of Applications On Performance

An application’s intrinsic memory access and reuse pattern can also affect its performance. In this section, we use the application characterizations from Table 4.1 to provide insights in making design decisions for our proposed approach.

![Reuse breakdown for CNF applications. It shows the percentage of addresses in L1 D-cache have been referenced by m number of times. The values of m are shown in the legend.](image)

In Table 4.1, the CI Category contains of six applications where enabling the L1 D-cache has no performance impact. We categorize the reasons behind such insensitivity into four typical scenarios: (1) workloads have no read access from the global memory, such as LEK and CFD; (2) workloads have the characteristics of intensive branching and very little memory access, such as MYC and GS, which also have very low IPC due to branch divergences; (3) streaming workloads, such as FFT, in which the global memory access happens only for loading the input signal sets into the shared memory; (4) workloads have high memory level parallelism, such as PF, where the critical path is determined by the first memory access miss. Because of their performance insensitivity to the L1 D-cache, the CI applications are not the focus of our study. However, we include them to prove the robustness of our proposed design.
Figure 4.4. The reuse distance histograms of L1 D-cache access stream of the CNF applications. The reuse distance is computed at cache access level instead of cache structure level (e.g. the reuse distance for access pattern ABCA is 2).

Figure 4.5. Hit rates and data reuse characteristics of GPU workloads: (a) Hit rates of CNF and CF applications; (b) The percentage of data accesses with and without data reuse.

The cache sensitive workloads include both CNF and CF applications. For CNF workloads, bypassing all memory requests from L1 D-cache improves the performance of NW, PTF, and SD2 by 24.2%, 36.8% and 36.6%, respectively. This is a direct result of the low cache
performance across the CNF workloads, as shown in Figure 4.5(a). Compared with the CF applications, the overall cache hit rate in CNF is much worse. For example, the hit rates for NW and HS are smaller than 1%. LUD has the highest hit rate in CNF (36.8%). However, 62.8% of its hits are write hits, which are useless due to the write evict policy in baseline GPUs [3]. On average, the cache hit rate for CNF workloads is only 8.9%, while the average for the CF workloads is as high as 55.4%. To understand why CNF workloads have such low cache performance, we evaluate the data locality of each application and quantify the data reuse rate of their memory stream in each SM. As shown in Figure 4.5(b), CNF workloads have a larger portion of data requests without any reuse, 53.2% on average compared to only 7.9% in the CF category. To analyze the characteristics of the reuse in the CNF workloads, we present the reuse distance histograms of these workloads in Figure 4.4. As shown in Figure 4.4, the reuse distances are relatively high when the cache block size is 128 bytes (baseline architecture). Such large reuse distances (>128) present a challenge for the limited number of blocks (i.e., limited capacity) in the L1 D-cache. Also, Figure 4.3 shows that memory requests for the majority of CNF applications (other than PTF and SSSP) are only reused less than three times. These data accesses with no or low reuse cause two problems: (1) they will contend with accesses with high reuse for resources and resulting in the average stall time for CNF workloads as high as 48.04% of the total execution time; (2) such accesses without any reuse are scattered inside the memory access stream, which indirectly increases the reuse distance of accesses with reuse. Finally, without accurate locality information, deciding which memory requests should enter L1 D-cache is difficult for applications with irregular access patterns (e.g. SSSP), which has been neglected by the previous studies [19][39].

### 4.4 DESIGN METHODOLOGY

The focus of this work is to design efficient bypassing for a GPU L1 D-cache that can dynamically divert memory requests based on reuse patterns. As shown in Figure 4.1, there are two design points available to implement such mechanism: an independent hardware component between the coalescing unit and L1 D-cache (option 1 in Figure 4.1), and the existing L1 D-cache integrated with the locality filtering capability (option 2 in Figure 4.1).
We choose 2 over 1 because both structures consist of entries that can be identified as tags (e.g. L1 tag store) so such redundancy in option 1 will increase access latency by performing additional tag checks per access. Also, option 2 will likely have smaller area and higher energy efficiency. Accordingly, we propose to decouple the tag and data stores of the existing L1 D-cache and integrate the locality filtering capability into the tag store through minor hardware expansions. In this way, we can leverage the management of the independent tag store to control which memory requests can allocate data lines in the data store. By taking advantage of the tag store’s smaller entry size (8~9 bytes) compared to that of the data store (128 bytes per line), additional entries in each set can be added in the tag store with lower overhead such that it can capture the locality information of a working set larger than the data store size. We name our new design “Decoupled L1D” and its diagram is shown in Figure 4.6.

Figure 4.6: Decoupled L1D cache design diagram.
4.4.1 Decoupled L1D: Structure

As shown in Figure 4.6, the tag and data store in the Decoupled L1D cache are independent structures although we choose to let them have the same number of sets to simplify the management. The decoupled tag store has expanded the original tag store with more entries in each set, and each tag entry has also been padded with more fields. In addition to the original fields, including the address tag, a status field, and a LRU counter, the new tag store entry now contains a Reference Count (RC) field and a Position field. The RC field (6-bit) holds the reference frequency (reuse) accumulated for the address. The Position field (2-bit) connects a tag store entry with a data line in the data store using a pointer to record the data line’s position. Once a data line in the data store is allocated, the corresponding tag store entry needs to set the Position field accordingly. The new data store in our Decoupled L1D has the same structure as the original one, except that each set has added one field named Free Line, indicating how many free lines are available for use in the set. This field will be updated upon a new data line allocation or an eviction.

![Operation ow chart of the Decoupled L1D.](image)

Figure 4.7. Operation ow chart of the Decoupled L1D.
4.4.2 Decoupled L1D: Operations

Instead of having uniform operations in the current L1 D-cache design, our Decoupled L1D enables separate sets of update and replacement logic for the tag and data stores. Also, we add a new cache request status bypass into the existing four statuses, i.e., hit, hit pending, miss, and reservation failed, of a L1 D-cache request. Figure 4.6 and 4.7 describe how the Decoupled L1D operates.

In the initial architectural status, each tag-store entry and data line is cleared (Position field is set to invalid). The Free Line field is initialized with the associativity size of each set in the data store. On a probe miss of the tag store, a new tag entry is allocated, and a bypass status is returned, as shown in Figure 4.7. A bypass will not trigger the cache miss handler but instead it will directly take the Bypass Path (Figure 4.6) without allocating a line in the data store. If there are no free tag entries in that set, the entry with the smallest RC is taken as the replacement victim. In other words, the LFU (Least Frequently Used) replacement policy is used for the decoupled tag store. Since the way to update RC can significantly impact the lifetime of a tag entry, we apply a customized dynamic aging scheme to the RC fields of the tag entries in the following scenarios: upon an allocation or eviction in the data store, the RC values of all the corresponding tag entries in that set (not including the allocated or evicted tag entry) are reduced by 1. Thus, the RC values of stale entries (i.e. entries with no future reference or long re-reference interval that still remain in the tag store) are reduced and eventually become small enough to be evicted out of the tag store.

On a probe hit in the decoupled tag store, there are two different operations depending on whether the Position field has pointed to an allocated data line. If the field has been set with a data-store line position, this memory request will proceed as a cache hit or cache hit-pending, similar to the original L1 D-cache. Otherwise, it will increase the RC value of the tag entry and then compare it with a locality threshold, which is pre-defined based on workload characteristics so that the data blocks with no or low reuse will not be inserted into the L1 D-cache. The subsequent program flow after the comparison is shown in Figure 4.7. If there are no free data lines in a selected set upon a data-line allocation, the data store’s replacement
policy (e.g. LRU, SHiP, RRIP, etc) will be triggered to find the victim line for eviction. Then, the RC value of the corresponding tag entry is set to 0 and the rest of the tag entries in the set will be aged. This ensures that the temporal locality of the data lines is preserved in our design by using the replacement policy from the original L1 D-cache. The associated parameters introduced in this section (including the number of tag entries, the set associativity, and the locality threshold) have direct impact on the area, power and performance of our design. We explore them in Section 4.6.2.

Fairness: For a fair system, the decision on allocating and replacing tag entries must closely match the data reuse frequency and distance. In our design, the locality of the program is preserved. If a data block has high reuse, its RC will continue to accumulate and eventually pass the threshold to allocate a new line in the data store for capturing subsequent reuses. Also, the state-of-the-art cache replacement policies (e.g. LRU, RRIP, etc) are used in the data store to help evict the tag entries (dynamic aging) that exhibit a distant re-reference interval caused by frequent bursts of references. In this way, the tag and data stores are circulated in a sustainable way with minimum number of stale tag entries generated.

4.4.3 SM Dueling

Our proposed design aims to improve performance of CNF applications. For CF applications, whose performance benefits greatly from utilizing the L1 D-cache, a locality threshold on RC may delay the data to be stored in the cache for reuse. Therefore, we can disable the RC checking for CF workloads by setting the locality threshold to zero, i.e., all accesses go to the L1 D-cache. For our decoupled tag store, the newly added tag entries can be power gated to reduce the energy overhead. To decide which mechanism to use for an application during execution, we employ a simple, but effective SM dueling technique [63] based on the insights from Figure 4.5a. Assume that we have a GPU with N SMs (N≥2). Initially, we assign SM 0 to use our new design and SM 1 to use the original L1 D-cache. The remaining SMs can choose to use either type. At a pre-defined time interval (see Table 4.3), SM 0 and SM 1 will compare their cache miss rates: if SM 0 has a higher cache miss rate than
SM 1, then all the SMs will start using the original L1 D-cache; otherwise all the SMs will enable the new design.

### Table 4.2. Baseline architecture configuration.

<table>
<thead>
<tr>
<th>Component</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMT Core (SM)</td>
<td>15 cores, SIMD width=32, 1.4GHz, 5-stage pipeline</td>
</tr>
<tr>
<td>Max/SM</td>
<td>1536 threads, 32768 registers, 48 warps, 32 MSHRs with 256 entries</td>
</tr>
<tr>
<td>L1 Cache/SM</td>
<td>16KB/core, 128B line, 4-way assoc, 1-cycle hit latency</td>
</tr>
<tr>
<td>Shared Mem/SM</td>
<td>48 KB; 32 banks; 3-cycle latency; 1 access per cycle</td>
</tr>
<tr>
<td>Unified L2 Cache</td>
<td>768 KB, 128KB/bank, 6 banks, 128B line, 16-way assoc</td>
</tr>
<tr>
<td>DRAM</td>
<td>6 memory channels, BW: 48 bytes/cycle, 1.4 GHz</td>
</tr>
<tr>
<td>DRAM Schedule Queue</td>
<td>Size = 16 and Out of order (FR-RCFS)</td>
</tr>
<tr>
<td>Warp Scheduling Policy</td>
<td>Greedy then oldest (GTO)</td>
</tr>
</tbody>
</table>

### 4.5 EXPERIMENTAL METHODOLOGY

Simulation Environment: Our proposed design is evaluated using GPGPU sim V3.2.2 [11], which is a widely used cycle-accurate simulator for GPU architecture research. The baseline architecture is modeled based on a generic NVIDIA Fermi GPU [3] and its configuration is shown in Table 4.2. Our new design can be similarly applied to other recent GPU architectures as well, such as NVIDIA Kepler and Maxwell. The design choices for our Decoupled L1D (based on a 16KBL1 D-cache) is shown in Table 4.3, which will be validated in Section 4.6.2 (design space exploration). Benchmarks: All 18 applications used in this study are shown in Table 4.1, which represents a wide range of optimized real-world GPU applications. Since shared memory is extensively used, the performance tends to be less sensitive to the L1 D-
cache, which makes them the perfect candidates to show whether our proposed design can still improve performance over optimized codes. We evaluate 14 workloads from Rodinia Benchmark suite with their default inputs. We include two additional applications: Matrix Multiplication (MM), a highly efficient version using tiled cache[57]; and Fast Fourier Transformation (FFT), an optimized version fully utilizing on-chip memory[88]. We also include two widely-used workloads: Barnes Hut N-body Simulation (BH) and Single-Source Shortest Paths (SSSP) from Lonestar GPU suite[15], both of which exhibit irregular memory access patterns. BH exhibits memory irregularity from data-dependent memory accesses, and SSSP has access irreuality from the memory traversing in a recursive iteration. In our experiments, all workloads run to completion on the simulator.

<table>
<thead>
<tr>
<th>Table 4.3. Configurations of our proposed Decoupled L1D.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Decoupled L1D</strong></td>
</tr>
<tr>
<td>#entries</td>
</tr>
<tr>
<td>structure</td>
</tr>
<tr>
<td>replacement policy</td>
</tr>
<tr>
<td>threshold</td>
</tr>
<tr>
<td><strong>SM Dueling</strong></td>
</tr>
</tbody>
</table>

4.6 RESULTS AND ANALYSIS

4.6.1 Overall Performance Evaluation

To evaluate the performance of our proposed Decoupled L1D cache design, we compare it with several related schemes and show the results and detailed analysis. As discussed previously, Decoupled L1D uses state-of-the-art replacement policies (e.g. LRU, RRIP, etc.) in the data store to preserve data lines’ temporal locality. To make a fair comparison, we use
the same cache replacement policy (a version of LRU implemented in GPGPUsim) as the basis for all the following schemes:

BL: Baseline Architecture. This configuration is shown in Table 4.2. All the global/local memory accesses will be inserted into the L1 D-cache. If resources are unavailable, the memory pipeline is stalled. This architecture favors CF workloads (Table 4.1).

BALL: Bypass all memory accesses from the L1 D-cache. All the global/local memory accesses will be diverted into the Bypass Path (Figure 4.1). L1 D-cache is disabled. BALL favors CNF workloads (Table 4.1).

Profiled-BYPASS: During a profiling run, we record the reference frequency for each data block by going through every coalesced memory request before they hit L1 D-cache. This profiled information is then used to direct fine-grained L1 bypassing (acted as a software filter) in a new run. This static approach is used to evaluate performance benefits that can be achieved when the reuse information of a workload is already available. Applying such an approach for bypassing has three major downsides: (1) it is dependent on the input and thread-configuration; (2) memory traces could be nondeterministic for each run; and (3) applying bypassing in a new run based on the previous profiling information may change the completion time of different thread blocks (TBs) and their associated dispatch order. We implemented three types of profiling-based filters: (1) a per SM local filter recording the intra-SM data reuse (local), (2) a kernel-level global filter recording the inter-SM data reuse (global), and (3) a combined local+global filter. All three filters have unlimited entries to store workloads’ memory addresses and their reference counts. Different locality thresholds from 1 to 16 were tested for the best performance. Among them, the global+local filter achieved the best average performance over BL. The reason is that with bypassing enabled, the thread block execution order is altered from the profiling run and the inter-SM reuses may become intra-SM reuse. From this point on, we will only use the global+local filter (named “Profiled-BYPASS”) for comparisons with other mechanisms. The profiling costs are not included in the figures of this section.
MRPB: The state-of-the-art GPU dynamic bypassing approach [39]. When any of the resource unavailable events happens that may lead to a pipeline stall, the memory requests are bypassed from the L1 D-cache until resources are available. MRPB includes memory request reordering queues, which are also modeled in our experiment.

PDP-best: The state-of-the-art CPU bypassing approach for LLCs, as discussed in Section 3.1. Both static and dynamic PDP approaches have been proposed in [28], in which the performance upper bound of PDP is achieved by the static PDP approach. Thus we compare our design to PDP-best, which is the best performed static PDP-bypass with the optimal PD specific to each workload (by exhaustively searching all possible PDs).

Decoupled L1D: Our design as described in Section 4.

Based on the experimental results, we make the following observations and analysis:

Figure 4.8. Performance comparisons of various cache management and bypassing schemes for the CNF workloads.

(a) **Performance Comparisons For CNF Workloads**: Figure 4.8 shows that for CNF workloads our proposed Decoupled L1D achieves the highest performance, up to 56.8% and
an average of 30.3% performance improvement (using the geometric mean) over BL. The performance comparisons among the different schemes highlights the importance of cache bypassing for GPUs. BL (all memory accesses go into the L1 D-cache) performs the worst because without any cache insertion management, memory requests with little to no reuse are diverted into the L1 D-cache, causing cache pollution and congesting the memory pipeline. BALL (all memory accesses bypass L1 D-cache) improves the performance for CNF workloads due to fewer memory pipeline stalls, however this mechanism completely disables the cache and overlooks data reuse in the memory access stream. As shown in Figure 4.5b, there are still over 40% of the data accesses that have reuse for the CNF workloads. MRPB improves CNF’s performance over BL by a geometric mean of 19.2%, only bypassing when a resource (cache lines, MSHR, miss queue entry, etc.) unavailable event occurs, in order to reduce resource contention. The disadvantage of MRPB is that it does not consider any data reuse pattern in the access stream. In other words, MRPB could still let data with no or low reuse be cached and bypass data with high reuse. In comparison, our proposed Decoupled L1D enables a fine-grained and effective bypassing based on reference count, which can better capture the data reuse in an access stream. Figure 4.8 shows that our design outperforms MRPB by up to 36% and an average of 11% for CNF workloads. On average, our dynamic approach also achieves better performance than the Profiled-BYPASS. For some CNF workloads like BH, we even observe an up to 11% performance improvement, without even counting the profiling overhead in Profiled-BYPASS. This proves that our dynamic method is more flexible and practical than the profiling-based bypass, without encountering profiling overheads, inaccurate requests diverting, and the potential nondeterministic effects.
Figure 4.9. Performance comparisons of various cache management and bypassing schemes for the CF workloads.

Comparing with PDP-best, for the irregular CNF workloads like BH and SSSP, our Decoupled L1D outperforms PDP-best by 8% and 11%. For PTF, which has long reuse distances and burst access patterns, Decoupled L1D outperforms PDP-best by 10.33%. For the regular CNF applications with very low reuse such as HS, NW, and LUD (Figure 4.3), our design performs as well as PDP-best. Based on [3], the primary reason for creating a GPU L1 D-cache is for improving the efficiency of irregular workloads processing, since shared memory cannot address the irregularities at runtime. We argue that the essential goal of our design is to explore the potential efficiency improvement for irregular workloads using bypassing. Unlike the hit-rate based PDP approaches constrained by the protection distance (discussed in Section 3.1), our Decoupled L1D monitors the dynamic reuse frequency and reuse distances using the decoupled tag store, which better captures the reuse patterns for irregular applications. Additionally, with the customized dynamic aging, stale tag entries in the tag store can be evicted out timely to avoid cache pollution.

(b) **Performance Comparisons For CF and CI Workloads:** CF applications, which have favorable caching behavior and regular control, can suffer greatly from inaccurate bypassing due to disrupted data locality. In Figure 4.9, significant performance loss compared to BL has
been observed for BALL (up to 77%), MRPB (up to 19%), and PDP-best (up to 30%). On the contrary, we do not observe performance loss for our design because SM Dueling determines if the new management scheme fits well with the workload pattern and dynamically turns bypassing on or off in all SMs (see Section 4.4.3). For CI workloads, as expected, our design along with other L1 D-cache management schemes has no effect on performance (with an average variance of 0.03% over the baseline performance).

![Reuse Distance Histogram](image)

**Figure 4.11.** The reuse distance histogram of L1 access stream of PTF (a case with long reuse distance shown in Figure 4) shaped by different designs. The reuse distance is computed at the cache access level instead of cache structure level (e.g., the reuse distance for access pattern ABCA is 2).

(c) **Hit Rate Improvement:** Figure 4.10a shows that Decoupled L1D can significantly improve the L1 D-cache hit rate for CNF workloads over BL by an average of 38.5%. Our design also outperforms the hit-rate based scheme PDP-best, with significant improvements for SD2 (71%), LUD (24%) and SSP (13%). This also confirms that hit rate is not directly correlated to performance (Figure 4.8) on GPUs due to warp interleaving and resource congestion. Meanwhile, MRPB only achieves a 17.4% average hit rate due to its coarse-grained bypassing and lack of consideration for data locality. Furthermore, both NW and HS
have negligible hit rates (<1%) across different mechanisms, caused by low reference counts (RC) for their memory requests (either 1 or 2 according to Figure 4.3).

(d) **Shaping Cache-Friendly Memory Access:** Using the CNF workload PTF as an example, Figure 4.11 shows that our design can shape a more cache-friendly memory access stream than other approaches. First, the data accesses that have a long reuse distance and therefore low reuse frequency will not be inserted into the L1 D-cache. As shown in Figure 4.11, the accesses with reuse distances of 512, 1024, and 2048 (such reuses require a cache with the capacity of 2048*the 128 cache line size = 256kB) will be bypassed in our design to avoid cache pollution. Second, our design can reduce the reuse distance of data accesses by filtering out unfriendly memory accesses. For instance, the percentage of the reuse distance ‘1’ has been significantly increased by our design and the overall reuse distance range is shrunk to better fit the limited cache capacity.

![Figure 4.12. Impact of cache replacement policies on performance of CNF workloads on GPUs.](image)

(e) **Alleviating Various Contention:** Through our Decoupled L1D, the footprint of the filtered L1 access stream for CNF applications is dramatically reduced by an average of 63.4% compared to the baseline. This in turn reduces L1 capacity and conflict contention. Since the
bypassed requests do not compete for L1 MSHRs and miss queues (Figure 4.1), the stall cycles due to reservation fails have also been significantly reduced by an average of 84.5% for CNF workloads.

(a) L1 D-cache performance (based on hit rate) for various designs. This is the average hit rate for the overall kernel lifetime.

(b) L2 cache accesses per thousand instructions for various designs, representing L2 level traffic
Figure 4.10. Cache performance (hit rate) and memory traffic caused by bypassing in various designs.

(f) **DRAM and L2 Level Traffic After Bypassing:** Figure 4.10b and 10c show the L2 and DRAM traffic after using various bypassing mechanisms. In Figure 4.10b, our dynamic bypassing mechanism encounters the least amount of traffic to L2 (least pressure to NoC). The same observation can be made for DRAM traffic, shown in Figure 4.10c. Also, for the two irregular applications BH and SSSP, DRAM traffic from PDP-best is much higher than our design, which can result in resource congestion. This makes applying PDP approaches on GPUs less attractive because it requires warp throttling techniques to accompany PDP bypassing (e.g., the design in [19]) to be effective for irregular workloads. However, warp throttling can significantly reduce parallelism and subsequently reduce overall performance, which our design does not suffer from.

(g) **Impact of Cache Replacement Policies On Performance:** Advanced replacement policies such as RRIP [38] and SHiP [82] have been proposed to amortize cache contention in CPU LLCs. Although RRIP and SHiP can reduce cache contention, the blocks with long reuse distances are still brought into the cache albeit in the LRU position. This is less a problem in CPU LLCs but is problematic for GPU L1 D-cache. Figure 4.12 shows the performance comparison of CNF workloads running on Baseline+LRU, Baseline+RRIP, and Decoupled
L1D+ LRU. We can easily observe that the replacement policies have little performance impact on CNF workloads. This is because unlike the CPU LLCs, GPU L1 D-cache has a much smaller cache associativity and capacity. Bringing in data with long reuse distances may evict more data with good locality. More importantly, without bypassing, all the L1 D-cache misses compete for limited cache resources (e.g. MSHRs and miss queue), resulting in serious resource congestion. Therefore, an efficient bypassing strategy such as ours is very necessary for GPUs, no matter which cache replacement policy is applied.

4.6.2 Design Space Exploration

Decoupled L1D tag store’s entry size: In the decoupled tag store, there are two types of entries based on their Position field (Figure 4.6 in Section 4.4). The entries with their Position set to valid are those that have already been connected with a data-store line position (named committed entries) and the rest are those that are competing for allocating a data line in the data store (named candidate entries). The lower bound number of the tag entries should be larger than the number of cache lines. Otherwise, some data cache lines are wasted. A larger size of tag entries can accommodate more candidate entries with a longer reuse distance. However, if the size of tag entries is too large, it will lose the sensitivity on reuse distance and cause cache pollution. For example, at the entry size 384, there are at least 256 entries that can be allocated for candidate entries since there are 128 cache lines (Table 4.2). If an entry has burst access pattern with reuse distance longer than 128, it can still stay in the tag store and may evict out other entries with short reuse distance by allocating lines in data store, causing cache pollution. Base on the results shown in Figure 4.13a, 256 entries should be used as the entry size in the tag store.
(a) The performance impact (IPC) of the entry number for the decoupled tag stores. All are normalized to the one with 256 entries. All are fully associative.

(b) The performance impact (IPC) of the set associativity of the decoupled tag store. The performance is normalized to the baseline 4-way associativity cache.
**Decoupled L1D tag store’s associativity:** Figure 4.13b presents the effect of varying the tag store’s associativity of the decoupled L1D on performance. The expanded tag store will record the reuse frequency of the incoming memory accesses and the size of way per set will affect how long an entry can reside in the tag store. When the number of ways is small, it may thrash the tag entries (that compete for allocating data lines) frequently due to high contention. When the number of ways is large, it can preserve the entries with a longer reuse distance. Our simulation results show that the 16-way design point can achieve only a 2% performance improvement over the 8-way with doubled size of the tag entries. Therefore, we select 8-way because it provides the highest performance per unit area.

**Reference Count (RC)’s locality threshold selection:** The RC field and locality threshold are discussed in Section 4.2 as part of our new tag store design. Figure 4.13c shows the effect of varying the values of the locality threshold on performance. The majority of our CNF benchmarks achieve the highest performance at the threshold value of 2 (or near the highest) due to their low data reuse pattern shown in Figure 4.3. Thus, we set the locality threshold value to 2 in our design. In this work, we did not implement an online system for determining
the value of the threshold for individual workloads because cache unfriendly workloads do not exhibit high variation for this design point.

Figure 4.14: The effectiveness of our design on the baseline L1 D-cache with different capacities. All data is normalized to a 16KB baseline cache.

4.6.3 Sensitivity to L1 D-cache Sizes

Figure 4.14 shows the sensitivity of our design on various L1 D-cache sizes. From the performance comparisons between the baseline caches and our approach, we have two observations: (1) For every cache size we studied, our design has shown performance improvements over the baseline, even at a larger cache size (128KB). With the increasing cache size, the performance improvements of our design over the baseline do not increase as rapidly. This is because smaller cache sizes are more sensitive to contention caused by large cache footprint while bigger caches can accommodate larger working sets and accesses with longer reuse distance. Also, for a workload, if the working set fits in the cache already, further increasing the cache size has no performance benefit. (2) Even a 16KB Decoupled L1D can achieve the average performance close to a 128KB baseline L1 D-cache, which indicates that
our design can significantly improve GPU performance without incurring higher area overhead.

Figure 4.15. Performance of our design with various warp scheduling policies, normalized to pure GTO performance.

4.6.4 Sensitivity to Warp Scheduling Policies

Prior sections use the Greedy-Then-Oldest (GTO) warp scheduling policy for experiments. Figure 4.15 shows the sensitivity of our design to various warp scheduling policies. Loosely Round Robin Scheduler (LRR) is oblivious to the access locality while GTO outperforms it by preserving the intra-warp data locality. The two-level scheduler (2LEVEL) proposed by Narasiman et al.[60] exploits inter-warp data locality by scheduling groups of warps (called FG) together to ensure different groups touching long-latency memory accesses at different times. However, 2LEVEL still performs worse than GTO. Rogers et al.[66] found that performing inter-FG in round-robin would destroy intra-warp locality of old warps, which however can be captured by GTO. Figure 4.15 highlights that improving the scheduling policy can increase performance by reducing the inter-warp contention. Our design complements it by further reducing the intra-warp contention (e.g., conflict misses & MSHR reservation fails),
which cannot be addressed by any warp scheduler. Therefore, our design can achieve further performance gains with different warp scheduling policies.

4.6.5 Hardware Cost And Energy Efficiency

We evaluate the hardware cost and energy efficiency of our Decoupled L1D design using CACTI 6.5 [81]. Hardware Cost: The major source of area overhead for supporting the SRAM-based Decoupled L1D comes from the expanded tag store. We didn’t change the bandwidth and number of ports in the Decoupled L1D and we assume 1 additional cycle for the extended tag-store probe. However, this additional latency is hidden by the thread-level parallelism, similar to the additional buffer latency described in [39]. Each tag has two additional fields, one for RC (6 bit) and the other for Position field (2 bit). With the expanded tag entries (256), the total area of Decoupled L1D is estimated as 0.244 mm$^2$ for the entire 15-SM system using 45nm technology, which is 0.008 mm$^2$ more than the original L1 D-cache. Such additional area represents approximately 0.02% of the GTX480 area[3], which is also a 15-SM system implemented in a 40nm technology. Other miscellaneous overheads such as the 2-bit per set ‘Free Lines’ counter in Decoupled L1D are negligible in comparison.

Energy Efficiency: Since our design is a simple tag store extension to the existing L1 D-cache, it does not require a large amount of energy-consuming cache logic circuitry. From the baseline L1 D-cache to Decoupled L1D, the dynamic read energy per access and total leakage power increases by 0.002 nJ and 1.7 mW respectively. Such small energy overhead can be disregarded compared to the substantial energy improvements from our design by significantly reducing L1 cache misses (by up to 80% for SSSP) and off-chip (DRAM) memory accesses. Specifically, the three CNF workloads with the most energy reduction are PTF (10.4%), BH (14.7%) and SSSP (11.4%).

4.7. RELATED WORK

CPU Cache Management: Prior studies have looked into optimizing LRU-based cache replacement policies for better performance, such as PiPP [83], RRIP [38] and SHiP [82]. However, due to the small cache capacity and massive parallelism in GPUs, even the most advanced replacement policies cannot address their cache contention problems. Our locality-
driven dynamic bypassing design aims to decide if a memory access should even be inserted into the cache, in order to reduce cache contention. Our approach is complementary to these cache replacement optimizations and in fact they are applied to preserve the temporal locality of the cache lines in our design. There also have been some work [28][44] proposed for CPUs on cache bypassing, which are usually designed for LLCs, where data locality is already filtered by previous level(s). But the poor locality of GPU workloads imposes difficulty for these approaches to make robust predictions for L1 D-cache (where locality is unfiltered). Our approach is specifically designed for throughput-oriented architectures like GPUs and it outperforms the state-of-the-art CPU bypassing approach PDP-best for both CNF and CF workloads, especially for irregular applications. V-Way[63] cache manipulates data store to enable global data replacement. On the contrary, our design focuses on integrating locality filter into the L1 D-cache for bypassing by using an expanded tag store to trace the reuse frequency of accesses, with significantly different structures and operations.

GPU cache management: Previous work [66][60] has made efforts on improving cache performance by changing the warp scheduling policies. For instance, CCWS [66] dynamically throttles active warps to avoid cross-warp contention by using a victim cache tag array. However, warp schedulers can neither address intra-warp contention nor eliminate cache pollution that is extensively caused by caching “unfriendly” data. Furthermore, warp throttling methods can reduce parallelism for processing memory requests and subsequently limit the resource utilization. MRPB [39] designs a FIFO queue for reordering memory requests in order to reduce intra- and inter-warp contention. It also uses a simple bypassing mechanism, which only bypasses when intra-warp contention occurs upon unavailable resources (e.g. MSHR reservation fails). More importantly, it does not consider any data locality of the access stream, which results in incorrectly bypassing accesses with good reuse. In contrast, our design aims to only store the data with high re-references and short reuse distance to maximally reduce cache pollution and resource contentation. As shown in Section 4.6.1, for both CNF and CF workloads, our design outperforms MRPB significantly, which outperforms CCWS [66] and [60]. Work [19] directly applies a dynamic PDP approach (runtime best-effort prediction using hardware sampling) on GPUs for L1 bypassing and uses warp throttling if contention at L2
and DRAM level are generated by PDP. Our efficient design outperforms PDP-best (exhaustively search for the best PD) for both CF and irregular CNF applications without increasing L2 and DRAM level traffic. Xie et al.[85]analyzed which global-memory loads should be cached at compile time and chose a subset of thread blocks to use the cache at runtime. In [73], the program counters (PCs) of memory instructions are used to make a cache-bypassing decision. In comparison, we rely on memory address to drive our dynamic cache bypassing. As GPU kernels are relatively short, the working set of a memory access instruction can be relatively large, especially considering the same PC in all the thread blocks. Therefore, the instruction-level bypassing [73] is more coarse-grain and may lose the opportunity of exploiting data reuse at memory address granularity. Compiler [84]and software level [18][70]techniques were also proposed to improve GPU cache performance. Compiler-driven mechanisms depend heavily on inputs and profiling runs. Although they can be effective to optimize cache performance for regular applications, they cannot predict the runtime behaviors of irregular applications. Unlike our dynamic hardware approach, software techniques [18] need to directly interact with applications for code optimization or data transformation, and can be detrimental to cross-platform performance portability[39].

4.8. CONCLUSIONS

In this paper, we analyze the GPU workloads to reveal the data reuse characteristics of different types of applications. For cache unfriendly (CNF) applications, their memory access stream feature accesses with low reuse and/or long reuse distances, causing severe cache contention and resource congestion. To address this challenge, we propose a locality-driven dynamic bypassing solution that integrates locality filtering functionality into the decoupled tag store of the current GPU L1 D-cache through simple and cost-effective hardware extensions. Experiment results show that our design achieves significant performance and energy improvements over the baseline caches and outperforms the state-of-the-art CPU and GPU cache bypassing schemes. It can significantly reduce various levels of contention without generating extra memory traffic and remain effective for various cache capacities and warp scheduling policies.
Chapter 5

Optimizing Memory Efficiency for Deep Convolutional Networks on GPUs

5.1 Introduction

The success of deep Convolutional Neural Networks (CNNs) in the 2012 ImageNet recognition competition (Alex-Net [45]) has made them as one of the most promising machine learning techniques. In the past few years, many deep neural networks have been developed and the latest CNN powered image recognition even outperformed human vision[35]. There are two main reasons for the success of deep CNNs. The first is large-scale training data sets and the second is large and deep neural network structures. Both require substantial computational and memory throughput. As a result, many-core processors like GPUs, featuring high computational throughput and memory access bandwidth, have become a popular accelerator for deep CNNs. Recently, a number of GPU-based accelerated CNN libraries have been developed. Cuda-convenet [48] was the first highly optimized CNN implementation on GPUs. After that, a number of popular machine learning frameworks such as Torch [25], Theano [13], Caffe [37] have released their own GPU libraries for CNNs. Among them, Caffe is the most popular deep learning framework and has been widely used in the machine learning...
community. The GPU hardware vendor, Nvidia, also develops a new library, cuDNN [22], which provides highly optimized and portable GPU kernel functions used in CNNs. Apart from them, there are also recent studies on accelerating CNNs by reducing the arithmetic complexity in convolutional layers [26][52][78], and using coarse-grain parallelism in thread mapping [74]. These existing works mostly focus on the computational efficiency of the network, especially that of convolutional layers. The memory efficiency of the network, however, has been largely overlooked. As deep neural networks have intricate data structure, the performance implication of memory behavior is not straightforward. Our study unveils that there are two aspects that have not been addressed yet pose non-trivial impact on memory efficiency and the overall CNN performance.

![Figure 5.1. Performance comparison between the CHWN layout (cuda-convnet2) and NCHW layout (cuDNNv4) on convolutional and pooling layers in AlexNet [45]](image)

The first one is data layout: As GPU thread organization, i.e., thread grid and thread block dimensions, is highly dependent upon data layout. Data layout determines the memory access pattern and has critical performance impact. For CNNs, the data are organized using multi-dimensional (i.e., 4 dimensions) arrays. Depending on how we place data into different dimensions, we have many ways (i.e., 24) to store the data in memory. While overlooked on previous work, surprisingly, we found that the data layout can significantly affect the performance and memory efficiency. Figure 5.1 shows the performance comparison of the two most popular data layouts on the AlexNet for different convolutional and pooling layers. From
the figure, we can observe that up to 6.9x layer-level performance improvement could be retained as a result of choosing a proper data layout. Moreover, even for the layers that have been always considered to be compute-bound, i.e., convolutional layers, we found that choosing the suitable data layout could lead up to 2.3x performance improvement. Since each dimension has distinct memory access patterns and the size of each dimension can also affect the performance, the performance impact from data layout is complex and difficult for developers to reason about. The problem is further complicated when considering different types of layers in a CNN as each type may also prefer different data layouts. As shown in Figure 5.1, the different data layouts in both convolutional layers and pooling layers yield non-trivial performance differences, and neither data layout can suite the best for all the layers. However, existing libraries only employ one data layout for all the CNN layers. Such a single and uniform data layout in the existing design mismatches the inherent heterogeneity in different layers used in a CNN.

The second one is redundant off-chip memory accesses. Our performance analysis shows that the memory efficiency of the memory-bounded pooling layers and classifier (i.e., softmax) layers is far from optimal due to the overlook on their off-chip memory data accesses. First, a CNN usually requires multiple steps to complete and there exists sequential data dependence across the steps. The common practice is to use a kernel for each step. However, it incurs high cost for inter-kernel data communication as the data pass through the bandwidth-limited off-chip memory. Second, leveraging data locality for high memory performance is an important optimization. However, how to optimize locality with different data layouts has not been addressed in existing CNN libraries.

In this study, we look into these memory issues and propose a set of methods to optimize memory efficiency for accelerating CNNs on GPUs. The main contributions of this work are:

• First, we characterize data layouts in various CNN layers, and reveal the performance impact of different layouts. Then we derive a light-weight heuristic to guide the data layout selection with minimal profiling overhead;
•Second, we support one network with multiple data layouts by proposing a fast multi-dimension data layout transformation on GPUs. We integrate the support for automatic data layout selection and transformation into a popular deep learning framework, Caffe.

•Third, we study the memory behavior of the memory-bounded pooling and softmax layers and optimize their memory access efficiency on GPUs.

•Finally, we perform rigorous evaluation and result analysis on different types of layers and representative networks, and demonstrate high performance improvements for both single layers, and complete networks.

With the promising results on the state-of-the-art networks including LeNet[53] and AlexNet[45], our work will hasten the development of deep neural network libraries on GPUs, hence contributing to the advances of machine learning applications.

5.2 Background

In this section, we first introduce the structure of a CNN, and summarize the algorithm characteristics for the major type of layers in CNNs. Then we describe the GPU-accelerated CNN libraries used in our study.

5.2.1 Convolutional Neural Networks

CNNs are a type of forward feeding Artificial Neural Networks (ANNs) inspired from animal visual cortex organization. An example of CNN is shown in Figure 5.2. As shown in the Figure, the intermediate results are different sets of feature maps. The working principle of CNN is to extract the local features from high-resolution feature maps and combine these features into more abstract low-resolution feature maps. These are realized by two alternating types of layers: convolutional and pooling layers. The last few layers are fully-connected classifiers that combine all local features together to produce the abstracted classification results. The detailed patterns of different layers are described below.
A Convolutional Layer extracts various features such as oriented edges, corners and crossings from input feature maps via convolutional filters, and then combine them into the more abstract output feature maps. The features in each feature map are 3D volume data with three dimensions: width, height and depth. With the large image data set and the massive computation power on GPUs, state-of-the-art CNN frameworks choose to process multiple images in a batch [45]. Thus, the input to a convolutional layer includes feature maps from multiple images and is organized as a four-dimensional (4D) array. The computation in the convolution stage is shown in Equation 1, where Ni is the batch size, Ci is the depth or number of input feature maps, Hi and Wi are the height and width of feature map, Fh and Fw represent the size of the convolution filter kernel, and Co is the output feature maps or the number of filters. The data volume in the convolution is four dimensional. To differentiate data layouts in the 4D arrays, we use the following notation in the paper: N (the number of images), C (the number of feature maps), H (the image height), and W (the image width). With this notation, we can see that Equation 1 uses the NCHW layout. In the NCHW data layout, the elements along the lowest dimension W are stored consecutively in memory. In comparison, the consecutive elements along the H dimension have a stride of W; the consecutive elements along the C dimension have a stride of in H*W; and so on.

\[
Out[Ni][Co][Hi][Wi] = \sum_{Ci=0}^{C} \sum_{f_h=0}^{F_H} \sum_{f_w=0}^{F_W} in[Ni][Ci][Hi + f_h][Wi + f_w] * filter[Co][Ci][f_h][f_w] \tag{1}
\]
Convolutional layers are typically most time consuming in the whole network. Therefore, achieving high arithmetic throughputs has been the main optimization goal [52][78]. Implementations using matrix multiplication and Fast Fourier Transform (FFT) have been proposed (see section III.A). Counter-intuitively, we observe that the convolutional layers are not necessarily only compute bound. Specifically, for convolutional layers with small C and N dimensions, the performance is actually memory bound similar to 2D convolution. Thus, their memory organization especially the data layout will have a substantial impact on their memory performance and overall performance.

A **Pooling Layer**, also called a down-sampling layer, summarizes the features from neighbors. In the example of Figure 5.2, each feature is scaled down by a sampling factor of 2x2. The pooling layer can perform different operation such as average or max. An average pooling layer can be defined as Equation 2, where X and Y define the size of the pooling window, and stride is the distance between successive pooling windows. If the stride is smaller than the window size, the pooling is performed in an overlapped manner.

\[
Out[Ni][Ci][Hi][Wi] = \left( \sum_{x=0}^{X} \sum_{y=0}^{Y} in[Ni][Ci][Hi \times stride + y][Wi \times stride + x] \right) / Y/X
\]  

(2)

Pooling layers are usually paired with convolutional layers in CNNs. Compared to convolutional layers, pooling layers have low arithmetic complexity, O(N*C*H*W). Its performance is mainly bounded by memory efficiency (i.e., bandwidth and latency).

A **Classifier (Softmax) Layer** is the final layer of a CNN for classification, which computes the possibility distribution over different labels. Before the softmax layer, there usually exist full-connected layers, which flatten the 4D feature maps into a 2D matrix. A standard matrix multiplication is used to implement a fully-connected layer [11]. The output will be fed into the softmax layer. The softmax layer will first find the maximal possibility over each batched image. Then, each possibility is shifted by the maximum. Next, an exponential operation is performed on each possibility. Last, all possibilities of each image are summed up and the summation is used to normalize the possibilities. The detailed algorithm is shown as the following five steps:
\[ Maxv[Nx] = \sum_{x=0}^{X} \sum_{y=0}^{Y} \max(In[Nx][Cy]) \] //Step 1

\[ Midv1[Nx][Cy] = \sum_{x=0}^{X} \sum_{y=0}^{Y} (In[Nx][Cy] - Maxv[Nx]) \] //Step 2

\[ Midv2[Nx][Cy] = \sum_{x=0}^{X} \sum_{y=0}^{Y} \exp(Midv1[Nx][Cy]) \] //Step 3

\[ Sumv[Nx] = \sum_{x=0}^{X} \sum_{y=0}^{Y} \text{sum}(Midv2[Nx][Cy]) \] //Step 4

\[ Out[Nx][Cy] = \sum_{x=0}^{X} \sum_{y=0}^{Y} (Midv2[Nx][Cy]/Sumv[Nx]) \] //Step 5

Each step in the softmax layer is element-wise matrix or matrix-vector computation. The low arithmetic intensity in these matrix vector operations, and the intermediate data communication across different steps also make it memory bound.

### 5.2.2 Deep CNN Libraries

Till now, there are a number of frameworks [9] developed for CNN research. Among them, Caffe[37], cuda-convnet[45] and cuDNN[22], are the most widely used and specifically optimized for GPU acceleration. In this paper, we study their memory efficiency for different types of layers. Caffe and cuDNN use the NCHW data layout for all the layers. There are two implementations for convolutions using this data layout. One is to use Matrix Multiplication (MM) to compute convolutions. This is the default approach as it can be used for different configurations of convolutional layers [22]. The other is based on FFT, which has been proposed and refined in recent works[59][78] and is integrated into the latest cuDNN version. Different from MM approach, FFT method requires significant overhead for intermediate data and works for certain types of convolution layers[78]. Different from Caffe and cuDNN, cuda-convnet chooses the CHWN data layout and uses the Direct Convolution method. For each library, we use their latest and best performing version, cuda-convnet2 and cuDNv4.

### 5.3 Methodology

In this section, we present our experimental methodology, including different types of benchmarking layers and networks, and the GPU hardware platform.

**Benchmarks**
The popular data sets for deep CNNs are MNIST [54], CIFAR [47], and ImageNet [65]. MNIST data set is used for hand-written character recognition. It contains 50,000 handwritten digit images in the training set and 10,000 examples in testing data set. LeNet [53] is the best studied network for MNIST. CIFAR10 contains the 10 different categories of objects such as cat, truck, airplane, etc and each category has 5000 training image and 1000 test images. We use the example Cifar network included in cuda-convnet for CIFAR10. ImageNet is a large-scale image collection with more than 1 million real-world images, which are classified into

<table>
<thead>
<tr>
<th>Layer</th>
<th>Ni</th>
<th>Co</th>
<th>H/W</th>
<th>Fw/Fh</th>
<th>Ci</th>
<th>S</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONV1 (CV1)</td>
<td>128</td>
<td>16</td>
<td>28</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>LeNet[53]; Model Error rate: 0.18% (epoch 200)</td>
</tr>
<tr>
<td>CONV2 (CV2)</td>
<td>128</td>
<td>16</td>
<td>14</td>
<td>5</td>
<td>16</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>POOL1 (PL1)</td>
<td>128</td>
<td>-</td>
<td>28</td>
<td>2</td>
<td>16</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>POOL2 (PL2)</td>
<td>128</td>
<td>-</td>
<td>14</td>
<td>2</td>
<td>16</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>CLASS1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>128 images and 10 categories</td>
</tr>
<tr>
<td>CONV3 (CV3)</td>
<td>128</td>
<td>64</td>
<td>24</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>Cifar10[48]; Model Error rate: 14.04% (epoch 100)</td>
</tr>
<tr>
<td>CONV4 (CV4)</td>
<td>128</td>
<td>64</td>
<td>12</td>
<td>3</td>
<td>64</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>POOL3 (PL3)</td>
<td>128</td>
<td>-</td>
<td>24</td>
<td>3</td>
<td>64</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>POOL4 (PL4)</td>
<td>128</td>
<td>-</td>
<td>12</td>
<td>3</td>
<td>64</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>CLASS2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>128 images and 10 categories</td>
</tr>
<tr>
<td>POOL5 (PL5)</td>
<td>128</td>
<td>-</td>
<td>55</td>
<td>3</td>
<td>96</td>
<td>2</td>
<td>ImageNet With AlexNet[45] Model</td>
</tr>
<tr>
<td>POOL6 (PL6)</td>
<td>128</td>
<td>-</td>
<td>27</td>
<td>3</td>
<td>192</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>POOL7 (PL7)</td>
<td>128</td>
<td>-</td>
<td>13</td>
<td>3</td>
<td>256</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>CLASS3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>128 images and 1000 categories</td>
</tr>
<tr>
<td>CONV5 (CV5)</td>
<td>64</td>
<td>96</td>
<td>224</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>ImageNet with ZFNet Model[90]</td>
</tr>
<tr>
<td>CONV6 (CV6)</td>
<td>64</td>
<td>256</td>
<td>55</td>
<td>5</td>
<td>96</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>CONV7 (CV7)</td>
<td>64</td>
<td>384</td>
<td>13</td>
<td>3</td>
<td>256</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CONV8 (CV8)</td>
<td>64</td>
<td>384</td>
<td>13</td>
<td>3</td>
<td>384</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>POOL8 (PL8)</td>
<td>64</td>
<td>-</td>
<td>110</td>
<td>3</td>
<td>96</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>POOL9 (PL9)</td>
<td>64</td>
<td>-</td>
<td>26</td>
<td>3</td>
<td>256</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>POOL10 (PL10)</td>
<td>64</td>
<td>-</td>
<td>13</td>
<td>3</td>
<td>256</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>CLASS4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>64 images and 1000 categories</td>
</tr>
<tr>
<td>CONV9 (CV9)</td>
<td>32</td>
<td>64</td>
<td>224</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>ImageNet with VGG Model [69]</td>
</tr>
<tr>
<td>CONV10 (CV10)</td>
<td>32</td>
<td>256</td>
<td>56</td>
<td>3</td>
<td>128</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CONV11 (CV11)</td>
<td>32</td>
<td>512</td>
<td>28</td>
<td>3</td>
<td>256</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CONV12 (CV12)</td>
<td>32</td>
<td>512</td>
<td>14</td>
<td>3</td>
<td>512</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CLASS5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>32 images and 1000 categories</td>
</tr>
</tbody>
</table>
over 1000 categories. The commonly used network includes AlexNet [45], ZFNet [90] and VGG [69]. The AlexNet is widely used as the baseline network for exploring various new networks. These five networks cover a wide spectrum of problem size (from 28 to 256 image size) and network sizes. Table 5.1 presents the configurations for different types of benchmarking layers selected from the five networks. In this table, the convolutional layers have different configurations with various batch input sizes and feature map sizes; and the pooling layers include both overlapped and non-overlapped configurations. For softmax layers, we test twelve different configurations used in various classification problems (See Section 5.6).

**Experiment setup.**

We conduct experiments on a Linux machine with Intel Xeon E5620 CPU. We use a Nvidia GTX Titan Black GPU for all our tests. It contains 6144MB device memory, 5121 GFLOPS computing capability and 235GB/s effective memory bandwidth[5].

**5.4 Memory Issue A: Data Layout**

In this section, we first characterize the performance impact of data layouts in different types of CNN layers and derive the heuristics of selecting the suitable data layout based on the size and type of a layer. Then, we develop a fast multi-dimensional data layout transformation to support different data layouts in one network. Finally, we present how to apply our flexible data layout support into popular libraries/frameworks.

**5.4.1 Data Layout in Convolutional Layers**

Convolutional layers are the most critical layers for CNNs due to their dominant time in the network. As described in Section 5.2.2, the implementation of a convolutional layer can be classified as general purpose ones including both the direct convolution and matrix multiplication method, and the special FFT method. We first look into the implication of data layout on general purpose ones. Then we analyze the effect of the FFT approach.

**Data Layouts in General Purpose Implementations**
For convolutional layers, we summarize their general properties from Table 5.1. First, the value of N as a batch size is generally a multiple of 16, and has limited choices as described in prior works [45][90]. Therefore, using N as the lowest dimension is a good choice to meet the requirements for coalesced memory accesses as the threads are organized accordingly. Given its limited choices, e.g., 32, 64 and 128, the optimization space is limited for this dimension. Second, although the width and height of each image typically are the same, the values can vary significantly for different convolutional layers, ranging from 12 to 224. Third, the depth of input feature maps (Ci) is 1 for grey-scale images or 3 for RGB images in the first convolutional layers, and then is a multiple of 16 in the rest convolutional layers, which can also provide regular memory accesses for a warp of GPU threads (warp size =32).

Based on the property discussed above on each dimension, combining the W and H dimension and using the dimension N as the lowest one are reasonable choices. This way, we have two candidate data layouts: CHWN or HWCN. The CHWN layout is used in cuda-convnet and each convolutional filter is applied on the H and W dimensions to generate one output feature map. A test of the HWCN layout shows the same performance as the CHWN layout on cuda-convnet because it doesn’t change the memory coalescing feature for the N dimension and keeps the same data reuse on the rest dimensions.

On the other hand, since the 2D convolution operations are applied on the H and W dimensions, a popular convolution implementation is to unroll the 4D matrix into a 2D array and perform computation in the form of matrix multiplication. The NCHW data layout is used. This is the strategy used in Caffe and cuDNN since matrix multiplication is a well-tuned algebra primitive available on virtually any platform with any input size (e.g., the cuBLAS library for GPUs). cuDNN also supports the NHWC data layout and our tests show that its NCHW layout outperforms its NHWC layout. The special FFT implementation in cuDNN also inherits the NCHW layout and we analyze its distinct effect later in this section. Here we use cuDNN to denote its default MM method. As Caffe also binds cuDNN in its implementation as an improved version, our main comparison is between the CHWN (cuda-convnet) and NCHW (cuDNN) data layouts.
Figure 5.3 shows the performance comparison between two different data layouts in the convolutional layers. As is discussed above, to accommodate the different underlying data layout, the implementations for the convolutional layer are also different, with direct convolution for CHWN, and MM (or FFT) for NCHW. Therefore, the performance impact from data layout is evaluated using the whole execution time of each layer. First, as shown in Figure 5.3, cuda-convnet outperforms cuDNN for the CONV1~CONV5 and CONV9 layers (by up to 6.5x speedup), but performs worse in the rest six convolutional layers. The differences between these two sets of layers lies in the N and C dimensions. Among the six layers performed better with cuda-convnet, CONV1, CONV3, CONV5 and CONV9 are the first layer in their corresponding CNNs, and the value of C of these layers is either 3 or 1. The layers, CONV2 and CONV4, also have a relatively small value of C, no larger than 64. For the rest layers performing better with cuDNN, the values of N are either 64 or 32. To further identify the sensitivities of data layouts on each dimension, we collect the results with one varying dimension size (N or C) and the other three being fixed.

![Figure 5.3. Performance comparison between two different data layouts for the convolutional layers in Table 5.1. The performance is normalized to cuda-convnet measured on GTX TITAN BLACK.](image)

Figure 5.4a shows the performance sensitivity when varying the value of N. We can see that cuda-convnet with the CHWN data layout is more sensitive than cuDNN as the value of
N changes. As shown in the figure, cuda-convnet outperforms cuDNN when the batch size $N$ is more than 64. With the underlying CHWN data layout, Cuda-convnet first allocates a warp of 32 threads in a TB to process 32 images such that the memory accesses are coalesced. In order to further reduce off-chip memory accesses, if the batch size $N$ is 128, cuda-convnet enables each thread to handle four images so that the data of these four images can be reused in the register file. If the number of images is less than 128, the reuse for images per thread would be reduced. As a result, the performance degrades quickly as the number of images is reduced. In other words, for the CHWN data layout, the $N$ dimension is used for both memory coalescing and data reuse (in registers), and therefore the performance is very sensitive to the value of $N$.

![Figure 5.4](image_url)

**Figure 5.4. Sensitivity study of data layouts on the $N$ and $C$ dimensions. CONV7 in Table 5.1 is used while others show similar trends.**
Compared to cuda-convnet, cuDNN and Caffe use the NCHW layout, and utilizes the cuBLAS [7] library (or internal routine) for matrix operations. Since a matrix multiplication has only two dimensions, a matrix unroll step (along H and W) is needed to expand the input matrix, and merge multiple dimensions into two dimensions [37]. Such matrix transformation overhead is more evident when the matrix size is limited. As a result, as shown in Figure 5.4b, when the value of C is less than 32, cuda-convnet performs much better as it doesn’t have the overhead of matrix expansion. On the other hand, cuDNN performs better when C is larger than 32 where matrix expansion leads to better data reuse and higher parallelism due to dimensions merging [22].

The performance implications from the data layout analysis above are two-fold. First, the N and C dimension are revealed as being highly correlated with memory performance. Second, a heuristic to select the suitable data layout for a convolutional shape can be derived based on the performance sensitivity analysis. For a given convolutional configuration, (1) if the value of C is smaller than a threshold Ct, CHWN will be preferred as the cost of memory transformation used in NCHW data layout is high; (2) if N is greater than or equal to a threshold Nt, the CHWN data layout is still the better choice as the value of N is large enough to achieve both memory coalescing and data reuse. For the rest configurations, NCHW is the preferred choice. Due to different memory designs, including cache capacity and memory bandwidth, on different GPUs, the thresholds (Ct and Nt) can vary. For example, on the experimented Titan Black GPUs, the (Ct, Nt) is (32,128). On a new GPU such as GTX Titan X GPU, (Ct, Nt) is (128, 64). Considering that the heuristic parameters only relate to the property of the hardware, for each GPU architecture, we only need one-time profiling (as the one shown in Figure 5.4) to determine these thresholds.

**Data Layouts in FFT-based Implementations**

Besides the implementation of convolution using direct computation and matrix multiplication, convolution can also be computed in the frequency domain using FFT since convolution in the space/time domain is equivalent as element-wise multiplication in the frequency domain. The FFT-based method for implementing convolution has three steps: 1)
Transform the input feature image and filter kernel into the frequency domain using FFT; 2) Perform element-wise product in the frequency domain; 3) Transform the result feature map from the frequency domain back into the space/time domain using inverse FFT.

For FFT-based approaches for convolution, additional memory is needed to pad the filter kernel to match the size of feature maps and this overhead can be large for small filters. One way to reduce such overhead is to use tiling (i.e., FFT-tiling). The NCHW data layout has been used in the FFT-based approach [22][78].

![Speedup Chart](image)

**Figure 5.5. Speedups of the FFT-based approach over the cuda-convets.**

The latest cuDNN library version 4 provides two options for the FFT method: FFT and FFT-Tiling. The FFT option computes the convolution as described above but requires a significant memory for data padding and intermediate results. The FFT-Tiling option also uses FFT but splits the inputs into 32x32 tiles such that the memory overhead can be reduced compared to the FFT option. Figure 5.5 shows the performance of various convolutional layers using FFT, FFT-Tiling and Matrix Multiplication (denoted as “cuDNN-FFT”, “cuDNN-FFT-T” and “cuDNN-MM”) with the NCHW layout compared to cuda-convnet with the CHWN data layout. For layers including CV5 and CV6, there are no results for both FFT options due to execution failures. For these layers, the required memory exceeds the hardware limit of 6GB on our GPU card. Among the layers that the FFT-based approaches can execute, there is no
clear winner between cnDNN-MM and the special cuDNN-FFT/cuDNN-Tiling method (they all use the NCHW layout). The FFT-based approach can perform better than cuDNN-MM when the filter kernel is large, the batch size is large or there are many channels such as CV7, CV10 layers. On the other hand, for small channel sizes, such as CV3, CV9, it performs much worse than MM method. In these cases, the overhead of multiple steps, i.e, multiple kernel launches, streaming memory in and out multiple times, and zero-padding to the input size, in the FFT approach can outweigh its algorithmic advantage. Regarding the impact of data layouts, we can see that the FFT implementations do not change our observations made from Figure 5.5: the CHWN layout is preferred in CV1~CV5 and CV9 while the NCHW layout is preferred in other layers. Our data layout heuristic remains effective in selecting the suitable data layout in each convolutional layer.

5.4.2 Data Layout in Pooling Layers

Compared to convolutional layers, pooling layers, another essential part of CNNs, are memory-intensive and also work on 4D data structures. Figure 5.6 shows the performance of pooling layers with different data layouts. As we can see, cuda-convnet (i.e., CHWN) significantly outperforms Caffe and cuDNN (i.e., NCHW) across the board, with a speedup up to 16.3x. For the pooling operation on the CHWN data layout, it works through each slice of feature maps in the 4D array and memory coalesced accesses can be achieved along the lowest N dimension. However, for the NCHW data layout, the way of memory accesses is different. As the H and W dimensions are in the lowest dimensions, the pooling operations on each pooling regions of the feature map are directly applied to the pixels that are stored in memory consecutively. As shown in Equation 2, to compute an output element, each thread will access a pooling window of input elements. Therefore, the consecutive threads in a warp generate memory accesses with a stride. Such strided accesses from a warp are un-coalesced, resulting in over-fetching and poor memory efficiency. For this reason, for pooling layers, their memory access pattern determines that the CHWN layout is always preferred compared to the NCHW data layout, as shown in Figure 5.6. When pooling is conducted in an overlapped way, the data layouts also impact on memory locality as there will be data reuse across overlapped pooling windows. Such locality impact is discussed in Section 5.5.
Figure 5.6. Performance comparison between different data layouts for the pooling layers in Table 5.1. The performance is normalized to cuda-convnet. The numbers on top denote the highest bandwidth (GB/S) achieved for each layer.

### 5.4.3 A Fast Data Layout Transformation for CNNs

From previous subsections, we reveal that a single data layout cannot satisfy the diverse layer configurations and layer types in a network. We also derive the preferred data layout based on the performance implication of the memory behavior of different layers. A subsequent question is how to enable the different suitable data layouts into one network? We propose an efficient data layout transformation. For brevity, we will mainly discuss the approach to transform from CHWN to NCHW.
Transforming an array in the CHWN layout to the NCHW layout is essentially a transpose operation on a 4D array. To implement parallel transpose for a 4D array on GPUs, a simple method is to construct a four dimensional thread mapping, and each thread dimension is used to handle a dimension of the array as shown in Figure 5.7a. The issue of this implementation is that the memory accesses of writing into the output array is not coalesced as the threads in a warp have a long stride of CxHxW when accessing memory, causing severe bandwidth

```c
1. __global__ void Transformation (float *in, float *out) {
2. //from CHWN to NHWC:
3. int tx =threadIdx.x, bx=blockIdx.x;
4. int by =blockIdx.y, bz=blockIdx.z;
5. out[((tx*gridDim.z+bz)*gridDim.y+by)*gridDim.x]+bx] =
6. in[((bz*gridDim.y+by)*gridDim.x)+bx]*blockDim.x+tx];}

Figure 5.7. Kernels for data layout transformation implementation

1. template <int N, int C>  
2. __global__ void OptTransformation (float2 *in,  
3. float *out) {
4. int tx =threadIdx.x, ty= threadIdx.y, bx =blockIdx.x,  
5. by=blockIdx.y;
6. //1 Matrix flatten 4D to 2D: [C][H][W][N]  
7. ->[C*H*W][N]  
8. int D2_W= N/2;   int D2_H =gridDim.y*gridDim.x*blockDim.x;  
9. //Shared Memory Tile for Subblock Transpose  
10. __shared__ float2 sh[C][33];     //Padding 1  
11. float2.
12. for (int i=0;i<N/64;i++) {  //handle 64 images every time
13.   int m =
14.   by*gridDim.x*blockDim.y+bx*blockDim.y+ty;
15. //2 Subgrouping in Shared Memory
16.   int D3_H = m/32; int D3_W = m % 32;
17.   int index  = D3_W + D3_H*32;
18.   sh[ty][tx] = in[index*D2_W+tx+i*32];  
19.   __syncthreads();
20. //3 Vector Transpose Index
21.   if(C%32==0) {
22.     out[(2*ty+i*64)*D2_H+(bx)*32+tx] = sh[tx][ty].x;  
23.   out[(2*ty+i*64)*D2_H+(bx)*32+tx] =
24.   sh[tx][ty].y;   }
25.   else if(C%16==0) {
26.     out[(2*ty+i*64)*D2_H+bx*32+tx] = sh[tx][ty].x;  
27.   out[(2*ty+i*64)*D2_H+bx*32+tx] =
28.   sh[tx][ty].y;  }
29.   else  {
30.     out[(2*ty+i*64)*D2_H+bx*32+tx] = sh[tx][ty].x;  
31.   out[(2*ty+i*64)*D2_H+bx*32+tx] =
32.   sh[tx][ty].y;  }
```

Naïve Kernel (a)

Optimized Kernel (b)
underutilization. To eliminate the un-coalesced memory accesses and achieve the optimized performance, we propose three optimizations as illustrated in Figure 5.7b.

First, we observe that among two data layouts NCHW and CHWN, three dimensions including C, H and W, have the same relative positions. Thus, we combine these three dimensions into a single dimension as CHW. Then NCHW becomes \([N][CxHxW]\), and CHWN becomes \([CxHxW][N]\) after combination. This way, we downgrade the 4D transformation into 2D data layout transformation by flattening the matrix such that a 2D thread hierarchy can handle it, as shown in Line 4~5 in Figure 5.7b. Then we can apply the shared memory-based Tile optimization [29] to achieve the coalesced memory access for global writes in Line 7~14. Additionally, in the Kepler architecture, shared memory has two bank modes: 4-byte accesses and 8-byte accesses. To fully utilize the bandwidth in 8-byte mode, we apply vectorization by grouping two consecutive float variables into a single vector type variable of float2 to form the larger tile. When writing-back the tile, it can be scattered into multiple consecutive rows based on the tile shape, with each vector variable writing in a coalesced manner as shown in line 16~24. This extra vectorization can further boost the bandwidth utilization as the global access transactions will be doubled for data fetching. It is applied when \(N\) is larger than 64.

5.4.4 Wrap Up: Automatic CNN Data Layout Support

Finally, we show that our data layout support can be easily integrated into existing CNN frameworks such as Caffe and cuda-convnet. In CNN frameworks, a configuration file defines a CNN by specifying a stack of various layers. Each layer is specified with the layer type and the size of this layer for the input, output data tensor and weight kernel. Applying our data layout support requires two changes. The first is to add a new field in each convolutional and pooling layer to indicate the data layout choice. By scanning through the network once, the field in each layer is set for which data layout is desired with only one-time profiling based on data layout analysis. The second is at the runtime of training or testing the network, at the completion time of one layer, an additional check is inserted to determine whether a data layout transformation is needed before passing the output to the next layer. By comparing the data
layout fields of the current layer and the next layer, if different, the transformation as discussed in Section 5.4.3 will be performed. We decide not to fuse data layout transformation into convolutional layers, because the different thread block configurations and memory access patterns lie between the convolution and 4D transpose kernels. The performance is adversely affected when they are combined. By wrapping up the data layout analysis and transformation, we enhance the CNNs framework with efficient data layout support based on the configuration and type of the layers of a network. This significantly relieves the burden of the machine learning developers from analyzing various layer configurations and further reasoning about their intricate GPU performance implications. The results in Section VI shows the effectiveness of our proposed support on various types of networks.

5.5 Memory Issue B: Off-chip Memory Accesses

In compliance with our efforts towards efficient memory access for CNNs, we revisit their inherent memory behavior. Specifically, our analysis shows that the memory performance of memory-bounded pooling and softmax layers is far from optimal. In this section, we look into their memory behaviors and propose effective optimizations to improve their memory efficiency.

5.5.1 Memory Analysis and Optimization on Pooling Layers

Figure 5.8. Overlapped pooling with a window size of 4. The 2D image is simplified to 1D vector.

In Figure 5.6, we report the highest bandwidth achieved in each pooling layer in the three libraries. As we can see, the bandwidth utilization is not high especially for the overlapped layers (i.e. when Fw>S) with a maximum of 173.9 GB/S and an average of 156.5 GB/S. For
Caffe and cuDNN, the average bandwidth is 52.3GB/S and 41.9GB/S, respectively. The reason for the poorly achieved bandwidth by Caffe or cuDNN compared to cuda-convnet is the data layout. As discussed in Section IV.B, the NCHW layout in the pooling layers leads to strided memory accesses, resulting in low access efficiency. Another common reason for the relatively low bandwidth utilization among all three libraries is the significant redundant data accesses. We illustrate it using a pooling operation on 12 consecutive elements in one dimension. As shown in Figure 5.8, the pooling window size is 4 and will slide with a stride of 2. Based on the pooling algorithm, each output element needs to load 4 input elements and totally 20 global memory accesses are required for the five outputs. Among these 20 global memory accesses, a number of them are redundant, as highlighted in the shaded ones in Figure 5.8. When the input is a 2D image, such redundant memory accesses will further increase.

To achieve high memory efficiency for the pooling layers, the first optimization is to use the CHWN data layout. Then, we leverage on-chip register file to enable data reuse so as to reduce the off-chip memory requests. In the pooling layer, the on-chip working set can be defined as the number of the output elements. Therefore, when we expand the working set per thread (aka thread fusion/merge/coarsening[87]), we increase the number of the output elements per thread. Then within a thread, the input elements used to compute these output elements can be cached in the register file and only need to be loaded from off-chip memory once. To find the best working set expansion factors along both directions, we design an auto-tuning process which aims to balance the register pressure and data reuse with a fine-grain search. In order to converge into the optimal version quickly, we apply a hill-climbing heuristic to prune the search space. With an initial factor of 2, the expansion factor continues to increase linearly if the performance improves. Otherwise it stops as further expansion leads to high register pressure thus limiting the TLP and resulting in lower performance.

5.5.2 Memory Analysis and Optimization on Softmax Layers

In Figure 5.13 (see section 5.6), we measured the highest bandwidth achieved for the softmax layers (the “BL_Best” bar) in existing libraries. As we can see, the overall memory performance is fairly low with the achieved highest bandwidth of 58.30 GB/s, less than 1/5 of
the peak bandwidth. There are two main reasons. First, for the softmax layers, to ensure the data dependence across steps, cuda-convnet and Caffe use a separate kernel for each step and five kernels are used in total to implement the layer. Between consecutive kernels, the intermediate results are streaming in and out of global memory, resulting costly off-chip memory accesses. Second, in each step, there are two loops: one covers the batch size N and the other covers all categories. Since the outer loop, which covers all batched images, has no loop-carried dependency, Caffe and cuda-convnet parallelize it by allocating one thread for each iteration. However, the inner loop with loop-carried dependence is not parallelized, which is used to perform the reduction type operation (Section 5.2.1) to compute the maximum or sum. The problem is that the parallelism of the outer loop is not enough for GPUs to hide instruction latency. If the number of images N is 128, very common for practical CNNs, the number of threads for the kernel is only 128.

To reduce the inter-kernel data communication cost, we propose to fuse all kernels into one such that the cross-step data communication can be promoted through registers or shared memory. For an efficient fusion, we need to consider three aspects: 1) different kernels may have different parallelism which needs to be coordinated for a uniform mapping into one kernel; 2) the synchronization and data dependence across steps need to be supported within the fused kernel; 3) cross-step data communication needs to be enabled through fast on-chip memory accesses.
To minimize the code change, we first identify that all five steps in the softmax layer have the same two-level loops, and the implementation using five kernels also have the same TB configuration after parallelizing the outer loop. Therefore, we can fuse these five kernels into a single kernel without modifying the TB configuration. Second, since the output of a step is used as the input of its next step, the communication between two kernels becomes the inner-thread communication and the data used for the communication can be promoted into register file or shared memory (line 3-12 in Figure 5.9). Thus, after kernel fusion, the intermediate global memory accesses are eliminated. Third, to address the problem of insufficient thread/memory-level parallelism, we propose to inject threads to further parallelize the inner loops. The inner loops of step 1 and 4 (Section 5.2.1) perform reduction operations, while the inner loops of the rest steps have no loop-carried dependences across loop iterations. Since the reduction can be parallelized using shared memory and synchronization within a TB, we can enhance the parallelism in all five inner loops. The optimized kernel for the softmax layers is shown in Figure 5.9.

As shown in Figure 5.9, the maximal value computed from step 1 can be stored in the shared memory and used in step 2 without using off-chip memory. Also, the input matrix elements can be loaded into shared memory or register in the first step and reused in the subsequent steps. The shared memory array, tmp_tile, is used as the intermediate temporary
array during parallel reduction and tmp_tile[0] keeps the computed maximal value. In step 2 (line 10~11), we can reuse the elements in both shared memory arrays (i.e., in_tile and tmp_tile) to perform the matrix subtraction operations. Moreover, the shared maximal value can be further reused in the per-thread register multiple times based on the multiple output elements each thread is to compute. As a result, inter-step memory communication is achieved now through the fast inter-thread shared memory coordination and intra-thread register reuse.

5.6 Results and Analysis

In this section, we first evaluate each optimization for the single layers described in Table 5.1, and then evaluate the impact on all five complete networks.

5.6.1 Results on Data Layout Optimization

First, we show that data layout has significant performance impact on convolutional layers and our heuristics presented in Section IV can find the suitable layout for all convolutional layers in Table 5.1. By measuring the best performance that can be achieved on each data layout, Figure 5.10 reports the performance differences, i.e., speedups, of the preferred data layout over the alternative (the bar labeled ‘Opt’). For example, on CV1, CHWN has a up to 6.5x speedup over NCHW; while on CV11, NCHW is the more suitable data layout, outperforming CHWN by 3.5x. On average, 2.48x speedup is achieved with the preferred data layout compared to use the alternative one. For the layers including CONV1, CONV2, CONV3, and CONV4, CHWN is the best layout as the value of N is 128. For the layers including CONV5 and CONV9, the number of input feature channels is less than 16. Thus, CHWN is still the best layout. For the rest layers, since the value of N is less than 128 and the value of C is more than 32, the NCHW layout achieves higher performance. Therefore, all the benchmarking layers in Table 5.1 confirm the effectiveness of our heuristics. We further examine our heuristics for various complete networks in Section VI.C.
Second, since consecutive layers in a network may have different preferred data layouts, the data layout transformation is needed and its overhead needs to be considered. Thus, we also evaluate the impact of the preferred data layout with the additional data layout transformation overhead. In Figure 5.10, the performance bar labeled ‘Opt+Naïve Transform’ shows the speedup of the preferred layout with the overhead of a naïve transformation to achieve it. The results labeled ‘Opt+Optimized Transform’ show the one with our optimized transformation. Their comparison highlights the impact of an efficient data layout transformation. For example, using the optimal data layout (i.e., CHWN) can provide 6.46x speedup for the CONV1. However, the overhead of the naïve transformation to achieve this data layout is large enough such that the data layout benefit is eliminated. With our optimized transformation, however, this layer still achieves as high as 4.02x speedup. The exceptions are CONV9 and CONV5, whose convolution filter is small and performance difference is minor (only 4.1%) among different data layouts. Therefore, even using optimized transformation, we cannot improve its performance. For these very small convolutions, our data layout transformation has very little negative impact as the layer itself has very minor impact on the
overall networks (less than 5%). Alternatively, to accurately determine whether transformation overhead will offset the performance improvement from achieving the suitable data layout, either the auto-tuning through one-time profiling or a performance model for analyzing the performance tradeoff between data layout impact and the transformation can be further explored. Overall, by considering the data layout transformation overhead on different layers, a naïve data layout transformation can not sustain the significant performance benefit from the suitable data layout and even degrade the overall performance. Using our proposed fast transformation to enable the optimal layout, an up to 4.02x (an average of 2.08x) speedup is still achieved.

Figure 5.11. Achieved memory bandwidth using three methods for data layout transformation. The Transform-Opt2 is not available for CV10, CV11, CV12 whose N is smaller than 64.

Figure 5.11 shows the detailed performance evaluation of our proposed data layout transformations. The bar of “Transform-Opt1” applies layout flattening and tiling with shared memory transpose (Section 5.4.3). It significantly improves the performance with an average of 6.48x speedup for all type of layers. By further applying the vectoring technique (labeled ‘Transform-Opt2’) on the applicable layers (i.e., for those with N is over 64), the achieved bandwidth has been improved to up to 14.7x, and an average speed-up of 7.5x. The optimized bandwidth for CONV6 has achieved of 229.5GB/S, which is 97.6% of the effective GPU bandwidth.
5.6.2 Results on Off-chip Memory Access Optimization

Figure 5.12 shows the performance comparison of different pooling layers between the existing implementations and our optimized kernels ‘Opt’ generated through auto-tuning. First, cuda-convnet outperforms the Caffe and cuDNN across the board, highlighting that the preferred data layout in pooling layers is CHWN. Second, with the preferred data layout of CHWN, for the overlapped pooling layer, our optimization on data locality labeled as ‘Opt’ can achieve higher performance with an average of 193.8GB/S memory bandwidth and improve the state-of-the-art performance by an average of 14.3%. This is the direct result of the significantly reduced global memory accesses through better data reuse. For example, in PL3 with a pool window of 3 and a stride of 2, our optimized kernel effectively reduced 9.1% global memory transactions and 36.0% DRAM accesses respectively, compared to cuda-convnet, and the overall performance has improved by 33.9%.

![Figure 5.12. Performance comparison among four different implementations for the pooling layers in Table 5.1. The performance is normalized to cuda-convnet.](image)

Figure 5.13 shows the memory bandwidth comparison between optimized and original kernels for the softmax layers. The bar ”BL_Best” shows the highest bandwidth achieved in existing libraries while the “Opt” shows the performance achieved using our optimized fused kernel. From the figure, we can see that our optimized versions consistently improve the memory bandwidth across all the softmax layers. For small layer sizes, the bandwidth can’t be well utilized. When the layer has large categories (such as 10000), the bandwidth achieved in
“Opt” can reach 220.95GB/S, which is 94.02% of the effective GPU memory bandwidth. In comparison, the highest bandwidth achieved in the BL_Best implementation (cuDNN) is still fairly low, 58.30GB/S, due to the inefficient memory access behavior. Among the optimizations, the efficient inter-step data communication enabled by our kernel fusion has contributed up to 3.53x speedup and an average of 2.81x speedup using the geometric mean. More threads from parallelizing the inner-loops can further bring an average speed up of 5.13x.

Figure 5.13. Performance comparison (GB/S) of softmax layers with a wide range of configurations. x/y means the batch size x and the number of categories y.

5.6.3 Results on Whole Networks

In this section, we evaluate the efficacy of our systematic memory efficiency optimizations on the whole networks. We use Caffe as our base framework and bind our two memory optimizations: flexible data layout support and optimized memory access for memory-bound pooling and softmax layers. The cuDNN library is also bound into Caffe, and runs as an improved version of Caffe. As cuDNN provides multiple implementation modes for the convolutional layer, each of them is performed to obtain the best performance. We name each of the evaluated mechanisms as follows:
cuDNN-MM: the convolutional layers use the standard matrix multiplication mode in cuDNN.

cuDNN-FFT: the convolutional layers use the FFT mode and falls back to the cuDNN-MM mode if failed.

cuDNN-FFT-T: the convolutional layers use the FFT-Tiling mode and falls back to the cuDNN-MM mode if failed.

cuDNN-Best: cherry-pick the best-performed/fastest one for each convolutional layer from all these available running modes.

Cuda-convnet: the network is running on cuda-convnet.

Opt: the network is running on the optimized framework bound with our proposed memory optimizations.

Figure 5.14 shows the overall execution time of the five complete CNNs using these different mechanisms. As we can see, with a single and uniform data layout in existing libraries, either cuDNN or cuda-convnet can only deliver the high performance for a subset of neural networks. For LeNet and Cifar, the performance of cuDNN is much worse than cuda-convnet, even using the best-possible performance from cuDNN-Best. On the other hand, cuda-convnet is significantly under-performed compared to cuDNN for AlexNet, ZFNet and VGG. This highlights the non-trivial performance drawbacks of employing fixed data layouts in current CNN libraries. By augmenting the flexible data layout and low-overhead data layout transformation, our optimized framework can achieve the highest performance for all types of networks. Compared to the state-of-the-art cuDNN library, considering the two most representative CNNs (LeNet and AlexNet), for LeNet, we can achieve 5.61x speedup over cuDNN-MM, 11.28x over cuDNN-FFT, 11.28x over cuDNN-FFT-T, and 5.61x over cuDNN-Best; for AlexNet, we can achieve 2.02x over cuDNN-MM, 1.17x over cuDNN-FFT, 2.87x over cuDNN-FFT-T and 1.16x speedup over cuDNN-Best.
Figure 5.14. The overall network performance comparison among various schemes.

Figure 5.15 shows the detailed performance comparison of different layers in AlexNet. AlexNet is the de-facto deep CNN structure for machine learning research. It contains 5 convolutional layers, 3 pooling layers, 1 softmax layer, and other layers such as normalization and full-connected layers. The convolutional layers in AlexNet have different configurations and no single data layout can be suitable for the whole network. For CV1, cuda-convnet performs better than cuDNN due to the suitable data layout of CHWN, while for the remaining four convolutional layers, cuDNN outperforms cuda-convnet significantly. Our optimized framework selects the right layout across all different convolutional shapes, CHWN for CV1 and NCHW for the rest. Then, for the pooling layers, based on our study, CHWN is the best and cuda-convnet consistently performs better than cuDNN. Our optimizations on these three pooling layers further improves the performance by up to 27.8% over cuda-convnet. Finally, for the softmax layer, our memory optimization shows the significant speedup, with up to 20.1x speedup over cuDNN and 8.2x over cuda-convnet. As there are four data layout transformations happening after PL1, CV2, CV3 and CV5, only minor overhead is incurred thanks to the efficient data layout transformation. Overall including the data layout transformation overheads, our optimized frameworks improve AlexNet by 1.46x over cuda-convnet, and 1.16x over cuDNN-Best (1.50x over cuda-convnet, and 1.19x over cuDNN-Best without the transformation overhead as shown in “Opt_NoTran” bar). The performance impact
of each layer on the whole network is different, with convolutional layer being the most performance dominant. Thus, achieving the flexible data layout for a network is the most critical optimization, contributing a 72% improvement. Comparatively, the off-chip memory access optimization contributes 28% due to the much smaller execution time of pooling and softmax layers.

![Figure 5.15. The performance comparison of different layers in AlexNet, Time Speedup Normalized to cuDNN-MM. “Opt_NoTran” bar denotes the performance without the transformation overhead.](image)

5.7 Related works

With the state-of-the-art recognition accuracy, deep CNNs have been applied into numerous application domains including image recognition, speech translation, drug discovery, etc. The pervasive usage of CNNs has also ignited the research on how to improve the speed of CNNs with increasingly larger data sets because the training time is still the limiting factor for applying CNNs. Specialized accelerators[20][21]have been proposed to build high-performance memory logics for CNNs. However, these accelerators are hard to program as they only support limited function units for a subset of layers. GPU clusters[24]
have been designed to achieve the high computational throughput. Their focus of performance optimization for multi-GPU CNNs [24][46] is how to efficiently partition the workload with low-overhead inter-device communication. Within each GPU node, the main execution blocks are still the basic layers as discussed in this paper.

To accelerate CNNs, GPU-based implementations for various types of layers such as convolutional, pooling, and softmax layers are available in various libraries [13][22][25][37][45] such as cuDNN and cuda-convnet. These GPU-based libraries relieve the significant burden of developing high performance CUDA code from machine learning researchers, and these GPU implementations have been widely used in the machine learning community. To accelerate the CNN performance on GPUs, recent efforts[52][78] focus on improving the computational efficiency of the convolutional layers while the data layout across the network remains unexplored. Cirensan et al. [23] implemented parallel convolution layers. Ren et al. [64] recently presented the vectorization of the basic layers. Marc [74] recently studied the coarse-grain parallelization using batch-level parallelism in CNNs. None focuses on the effect of the underlying data layouts and data access patterns for all these different CNN layers. To the best of our knowledge, this paper is the first study to look into the memory efficiency of accelerating CNN on GPUs. With the intricate data structure in CNNs, the performance implication of their memory behavior is still not clear. Our performance analysis unveils various performance implications and systematically optimize their memory efficiency, showing it to be a substantial performance factor for different types of CNNs.

We also observe that like FFT approach, more techniques leveraging arithmetic complexity theory might be proposed in the future for CNNs, e.g., the recent proposal from Nervana Systems[52]. They can set state-of-the-art speeds for a group of layers, for which they suit. Nevertheless, the underlying impact from data layout remains. The reason is that with compute efficiency being addressed with these new approaches, the performance impact of the memory efficiency is likely to become more important.
5.8 Conclusion

This work looks into the memory efficiency problems in current GPU-accelerated deep CNN implementations, including both data layouts and off-chip memory accesses. Our detailed study unveils the impact of data layouts on different types of CNN layers and their performance implications, then we propose the efficient data layout support as our solution. We further look into the memory access patterns of the memory-bounded layers, and propose effective optimizations to substantially reduce their off-chip memory requests and inter-kernel communication. The experiments demonstrate the effectiveness of our memory optimizations and their universal effects on different types of layers and various complete networks.
Chapter 6

Instruction Cache Study for GPU Architecture

6.1 Introduction

Throughput processors such as GPUs employ the single-instruction multiple-thread (SIMT) execution paradigm by fetching and decoding one instruction for a group of threads (called warp) to amortize the instruction fetch and decode overhead. To explore the instruction reuse in the dynamic warp execution, GPUs dedicate on-chip L1 Instruction Caches (I_Caches) to provide the fast low-latency instruction access. As the instruction access stream has a very good locality, the L1 I-Caches can enjoy very high cache performance. However, with GPUs extending their usage into more diverse and general purpose applications, irregular and complex applications which have many execution steps are programmed on GPUs. Also, more highly optimized product code that leverage different optimizing techniques have been common. These makes GPU programs evolve with the large kernel size, leading to the high number of executable instructions. Due to the limited capacity of the L1 I-caches, the large instruction footprint causes serious cache thrashing. Furthermore, with GPU’s computation power increasing quickly, concurrent kernel executions have been common on GPUs to further
utilize the high number of computational and memory resources. Concurrent kernel execution, however, will further increase the instruction cache contention as the instruction footprint from concurrent-kernel execution will be even larger, disrupting the instruction cache locality and resulting in the low cache efficacy.

In this work, we look into the instruction cache efficiency for different types of GPGPU workloads in both single-kernel execution and concurrent-kernel execution. For a number of single-kernel and concurrent-kernel workloads, they have a large dynamic instruction footprint, making the small capacity of the on-chip GPU instruction cache well a performance bottleneck. In this study, we identify the instruction inefficiency and design efficient hardware prefetching mechanisms to substantially reduce the instruction cache misses. In our design, we leverage two hardware prefetching schemes to achieve high instruction cache efficiency. The first one is the light-weight next-line prefetching which fetches the subsequent instruction cache line into the instruction cache when the current instruction request results in a cache miss. The second one is to utilize a independent prefetching buffer that works for the instruction cache. Instead of contending the resources in the original cache, a number of streaming prefetching instructions will first be filled into the dedicated buffer and one instruction is promoted into instruction cache only when it is referenced. Experiment results show that our hardware prefetching mechanisms can enhanced the performance of GPU applications significantly, especially for the workloads with high instruction cache contention. For the next-line prefetching, it can achieve up to 1.42x speedup and an average of 1.25x speedup over the baseline instruction cache. For the prefetching buffer, it can achieve up to 2.14x speedup and an average of 1.48x speedup over the baseline instruction cache.

6.2 Background

Instruction Request Handling in GPU Architecture.

To handle the instruction request in GPU hardware execution pipeline, the fetch stage will execute by selecting an active warp to fetch the instructions from instruction cache. A warp cannot be selected if there is no empty slot in its instruction buffer or the warp is pending on the instruction cache misses. Once a warp is selected, the current program count is obtained
and the memory request for the next two instructions [11] will be issued into the instruction caches. If the cache probe returns to be a cache miss, a flag for this warp will be set by indicating it is pending on the instruction cache miss. On the other hand, if it happens to be the cache hit, then the fetch buffer will be filled with 2 instructions. In the decode stage, the fetch buffer will be checked for any valid instructions. If there are valid instructions in fetch buffer, the instructions will be decoded and sent into the instruction buffer based on its hardware warp identifier. In the issue stage, the warp schedule will select one ready warp, read the instruction from its instruction buffer and check the dependency. If one instruction can be issued, the instruction slot will be freed from the instruction buffer. In the execution stage, the instruction request is handled in the instruction cache. If the instruction cache misses happen, the miss handling logic will check the MSHR to see if the same request is also pending on the prior ones. If so, this request will be merged into the same entry and no new request will be issued. Otherwise, a new MSHR entry will be allocated for this request and a new cache line will be reserved for it. Meantime, the longer latency memory access will be incurred as it will be diverted into the next-level memory hierarchy.

**Prefetching for GPUs.**

Prefetching techniques have been the important architectural optimizations for CPU architecture[75]. Recently, there are more and more researchers looking at the memory prefetching mechanisms for GPUs[50][68][62]. The most recent and related one is [62]. As revealed in [62], instruction fetch performance has a non-trivial impact on overall application performances. Their work employs the light-weight next-line prefetching to reduce the instruction fetch stalls. In our work, we further study the advanced prefetching buffer mechanism for GPUs. Also, besides their focus on single-kernel execution, we also study the efficiency of instruction cache prefetching mechanisms for concurrent kernel executions.

**6.3 Methodology**

To evaluate the efficacy of the instruction cache, we model the generic FERMI-like [3] architecture using GPGPUsim V3.2.2[11], a cycle-accurate microarchitecture simulator. The following table shows the hardware configuration for the baseline architecture.
Table 6.1 Baseline Architecture Configurations.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td># of execution cores (SMs)</td>
<td>15</td>
</tr>
<tr>
<td>SIMD Pipeline Width</td>
<td>32</td>
</tr>
<tr>
<td>Number of Threads/SM</td>
<td>1536</td>
</tr>
<tr>
<td>Number of Registers/SM</td>
<td>32768</td>
</tr>
<tr>
<td>Shared Memory /SM</td>
<td>48kB: 32 banks; 3-cycle latency; 1 access per cycle</td>
</tr>
<tr>
<td>L1 Instruction cache/SM</td>
<td>16KB/core, 128B line, 4-way assoc, 1-cycle hit latency. Each instruction is 64-bit (8 bytes).</td>
</tr>
<tr>
<td>L1 Data cache/SM</td>
<td>16kB: 128B line, 4-way assoc 1-cycle hit latency</td>
</tr>
<tr>
<td>Constant Cache Size /SM</td>
<td>8k</td>
</tr>
<tr>
<td>Texture Cache Size/SM</td>
<td>12k, 128B line, 24-way assoc</td>
</tr>
<tr>
<td>L2 Data cache</td>
<td>768k: 128B line, 16-way assoc</td>
</tr>
<tr>
<td>Number of Memory Channels</td>
<td>6</td>
</tr>
<tr>
<td>Memory Channel Bandwidth</td>
<td>8 Bytes/Cycle</td>
</tr>
<tr>
<td>DRAM clock</td>
<td>1400 MHz</td>
</tr>
<tr>
<td>DRAM Schedule Queue Size</td>
<td>16, Out of Order (FR-RCFS)</td>
</tr>
<tr>
<td>Warp Scheduling Policy</td>
<td>Greedy then oldest scheduler</td>
</tr>
</tbody>
</table>

To evaluate the concurrent kernel execution, we also modify the GPGPU sim to support the multi-kernel launch and execution in each SM. As streams are supported in GPUs with the asynchronous, concurrent execution. We bind each concurrent kernel into one separate stream so that the kernels from multiple streams can be selected and dispatched into each SM. Thread blocks from different concurrent kernels may have different memory resource requirements. Thus, how to manage the resource allocation and deallocation for concurrent thread blocks from different kernel is also required. In order to reduce the resource fragmentation in the dynamic execution, we implement the resource management in this way: the thread blocks from concurrent kernels are first launched in a round-robin way as long as the remaining resource can meet their allocation requirement. Then in the dynamic execution, if one thread block completes its execution and exits from the SM, the thread block dispatcher will first select another waiting thread blocks in the same kernel. If the kernel has finished execution thus no available thread blocks to be dispatched, the thread block dispatcher will select a thread block from other kernels that can be launched based on the available resources.
We examine a wide range of benchmarks from various benchmark suites including CUDA SDK[5], Rodinia[17], GPGPUsim suites[11], Lonestar GPU[15]. We also evaluate the GPU kernels that have been highly optimized including the basic algorithm of FFT [88] and MM [58]. Table 6.2 shows the 14 benchmarks that are used in our studies. These benchmarks represent a wide range of optimized real-word GPU applications with different kernel size from the very small kernels (such as PF) to large and complex ones (such as WP). Besides, we also craft four concurrent kernel workloads to evaluate the GPU instruction cache performance on concurrent kernel execution.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Threads</th>
<th>Blocks</th>
<th>Registers/Threads</th>
<th>Shared mem(KB)/TB</th>
<th>Active TBs/SM</th>
</tr>
</thead>
<tbody>
<tr>
<td>PF</td>
<td>(256,1,1)</td>
<td>(463,1,1)</td>
<td>16</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>BP</td>
<td>(16,16,1)</td>
<td>(256,1,1)</td>
<td>11</td>
<td>1.0624</td>
<td>6</td>
</tr>
<tr>
<td>NN</td>
<td>(256,1,1)</td>
<td>(168,1,1)</td>
<td>6</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>B+Tree</td>
<td>(256,1,1)</td>
<td>(6000,1,1)</td>
<td>23</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>RAY</td>
<td>(16,8,1)</td>
<td>(16,32,1)</td>
<td>45</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>HIST</td>
<td>(128,1,1)</td>
<td>(1024,1,1)</td>
<td>29</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>MM</td>
<td>(8,16,1)</td>
<td>(8,32,1)</td>
<td>63</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>FFT</td>
<td>(128,1,1)</td>
<td>(1024,1,1)</td>
<td>56</td>
<td>8.5</td>
<td>4</td>
</tr>
<tr>
<td>STO</td>
<td>(128,1,1)</td>
<td>(384,1,1)</td>
<td>47</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>WP</td>
<td>(16,8,1)</td>
<td>(5,8,1)</td>
<td>63</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>FFT-HIST</td>
<td>(128,1,1)</td>
<td>(1024,1,1)</td>
<td>56.29</td>
<td>8.85</td>
<td>3</td>
</tr>
<tr>
<td>STO-HIST</td>
<td>(128,1,1)</td>
<td>(384,1,1)</td>
<td>47.29</td>
<td>16.16</td>
<td>3</td>
</tr>
<tr>
<td>FFT-MM</td>
<td>(128,1,1)</td>
<td>(1024,1,1)</td>
<td>56.63</td>
<td>16.8</td>
<td>4</td>
</tr>
<tr>
<td>STO-FFT</td>
<td>(128,1,1)</td>
<td>(384,1,1)</td>
<td>47.46</td>
<td>16.8</td>
<td>4</td>
</tr>
</tbody>
</table>

6.4 GPU Instruction Cache Efficiency Analysis

To reduce the instruction cache contention, we apply the most direct optimization by increasing the capacity and associativity of the instruction cache and observe how it effects the overall performance. These optimizations are expensive and impractical to implement in real GPU cache due to increased access latency, area, and power consumption. Figure 6.1 shows
the performance comparison for different workloads on different cache configurations. As we can see, compared to the workloads in the left-hand side, the workloads in the right side are more performance sensitive to the capacity and associativity of the instruction cache, which indicates instruction cache is a performance bottleneck for their performance. For these workloads whose performances is not sensitive to the instruction cache, we name it as Low Cache Contention workload (LCC). The other workloads whose performance are bounded by the instruction cache are High Cache Contention (HCC) workloads. Inside the HCC, we further observe that the instruction cache capacity with 32KB can fit well for the single-kernel workload. While for the concurrent kernels, the performance will be much better with a larger capacity such as 64KB or 2MB. This shows the concurrent kernels have a typically large instruction working set and thus much higher cache contention in 16KB baseline cache.

![Figure 6.1. The performance comparison for different cache configurations on various GPGPU workloads.](image)

To understand the instruction cache behavior, we further study the cache miss distribution in the dynamic execution time. We use FFT as an example to illustrate the run-time analysis.
Figure 6.2 plots the dynamic cache misses along the whole execution time for GPU with both 16KB cache and 32KB caches. The x-axis shows the execution time line with each time interval of 500 cycles. The y-axis shows the instantaneous cache misses in each time interval. As we can see, the whole execution process consists of a number of execution waves. In the first wave, both the execution on 16KB and 32KB instruction cache have the similar cache miss counts. Such behavior is reasonable as both types of caches will experience the compulsory miss, and these memory misses have also been distributed across the execution process of the wave. In the second wave, there is no dynamic cache miss with a 32KB cache and the instruction fetch request in this wave will be served well in the instruction cache, which is also the case for the subsequent execution. However, on the 16KB cache, it still experience cache miss in the second wave and the cache miss behavior is very similar to the first one (compulsory miss). This shows the small cache capacity cannot accommodate the instruction footprint of the running kernels, and the large instruction footprint lead to serious cache thrashing in each subsequent wave of the execution.

![Figure 6.2. The dynamic cache misses for FFT in the execution lifetime on 16KB instruction cache and 32 KB instruction cache. (Each unit is 500 cycles)](image-url)
6.5 Memory Prefetcher

To reduce the instruction cache miss and hide the long instruction latency for HCC workloads, we design efficient memory prefetching for instruction cache. There are two design options for the efficient prefetching. The first one is to simply prefetch instruction directly into L1 I-caches. Whenever the current instruction request is a cache miss, the next instruction cache line will be prefetched, aka. next-line prefetch. The prefetching instruction requests will be handled in the similar way as the regular cache request, except that on a cache hit or cache block eviction of the prefetching cache line, it will not update the LRU counter of the cache lines in that cache set.

The second one is to prefetch the multiple cache lines into a prefetching buffer. The prefetching buffer contains N number of buffers and each buffer contains M number of entries. Each entry includes the tag and instruction info field while each buffer has its LRU counter. The number of buffers N and the number of entries M can be configured. The prefetching buffer will be an independent component between the instruction cache and the next level memory. On an instruction cache miss, the prefetching buffer will be probed. If it is a hit instruction buffer, depending on the entry is reserved or valid, the request will return cache hit or cache hit pending. On the buffer miss, it will select a victim buffer based on the LRU counter, and issue M number of consecutive cache line requests. The prefetching cache lines will be filled into the buffer when the response instructions return. The LRU field will be updated for each buffer.

6.6 Experimental Results

Figure 6.3 shows the overall performance for HCC workloads on 16KB baseline cache and 16KB baseline cache with our next-line prefetching. As we can see, the next-line prefetching can effectively improve the performance. For the single kernel such as FFT, it can achieve 1.39x speedup over the baseline cache while for the concurrent kernels such as FFT-STO, it can achieve up to 1.42x speedup. On average, 1.25x speedup can be achieved for various HCC workloads. The performance improvement comes from the significant reduced cache misses.
For example, on 16KB baseline cache, for FFT kernel, there are 54180 cache misses while with next-line prefetching, the number of cache miss has reduced to 27956.

Figure 6.3. The performance comparison for HCC workloads on baseline cache and the baseline cache with next-line prefetching.

Figure 6.4 shows the performance improvement for HCC workloads on 16KB base line with prefetching buffer. As we can see, prefetching buffer can achieve the highest performance across the board. For STO_FFT, it can achieve a speedup of 2.14x. On average, 1.48x speedup is achieved over the baseline cache. Prefetching buffer also outperforms the simple next-line prefetching significantly by an average of 18.8%. This is because instead of directly competing on the limited cache resources (cache lines, MSHR, miss queue entries, etc), the prefetching cache line will be first filled into the buffer, with less resource contention. Furthermore, the multiple streaming requests on a cache miss also give a large future reuse window, which can further reduce the instruction cache misses in the execution. Considering the overhead of area and performance improvement, the best configuration for single-kernel execution is the instruction buffer with 2 buffers and 4 entries per buffer, while concurrent kernel execution favors 4 buffers with 4 entries per buffer.
6.7 Summary

To provide fast instruction cache access, L1 I-caches is one of the important on-chip resources to improve the memory efficiency for GPU architecture. This study looks into the efficacy of instruction cache on both single kernel and concurrent kernel executions. For the high cache contention workloads, the large instruction footprint lead to serious cache contention and cache thrashing, leading to low cache efficiency. To reduce the instruction cache misses, we implement efficient memory prefetching mechanisms including next-line prefetching and prefetching buffer for GPU instruction cache. The experimental results show that next-line prefetching can lead up to 1.39x speedup for single-kernel execution and 1.42x speedup for concurrent kernel execution. While the prefetching buffer can lead up to 1.96x speedup for single-kernel execution and 2.14x speedup for concurrent kernel execution.

Figure 6.4. The performance comparison between three cache design: the 16KB baseline cache, the 16KB baseline cache with next-line prefetching, and the 16KB baseline cache with prefetching buffer.
Chapter 7

Conclusion

In this dissertation, we study various performance-critical memory resources in GPU architecture and propose optimizations to optimize memory efficiency for GPUs in a whole stack, from micro-architecture, compiler to application-level algorithms. We first present our in-depth study on understanding the tradeoffs of two commonly-used on-chip caches including software-managed caches and hardware-managed caches in GPU architecture. We reveal many interesting and somehow unexpected tradeoffs among their usage with four detailed case studies on GPU architecture.

Secondly, due to the intricate memory hierarchy in GPUs, explicitly managing these different memory resources is non-trivial for even skilled programmers. As these on-chip memory resources also vary across different generations, performance portability becomes a daunting challenge. To address them, we present the novel compiler algorithm by efficiently revising the memory types of these data into these different on-chip memory resources in an optimized way to achieve both performance enhancement and performance portability.

Thirdly, on-chip L1 D-caches are critical memory resources to provide high-bandwidth and low-latency memory access without programming the shared memory. However, we observe that the high number of memory requests from single-instruction multiple-thread (SIMT) cores
makes the limited capacity of L1 D-caches a performance and energy bottleneck, especially for memory-intensive applications. We make the key observation that the memory access streams to L1 D-caches for many applications contains a significant of requests with low reuse, which greatly reduce the cache efficacy. And we propose locality-driven dynamic cache bypassing as our solution that integrates the locality filter functionality into the decoupled tag store of the current GPU L1 D-caches through simple and cost-effective hardware extensions.

Fourthly, deep convolutional neural networks are critical applications/algorithms on GPU architecture. With the previous works on studying GPU memory hierarchy, we detailed look into the memory access efficiency for accelerating CNNs on GPUs. Our detailed study shows that there are two memory efficiency problems including the data layout and memory access pattern that have not been addressed yet pose a non-trivial performance impact on the overall network performances. We unveil the performance implications of data layout on different types of network layers and also propose effective optimizations to substantially reduce the off-chip memory requests and inter-kernel communications. Experimental results demonstrate the effectiveness of our memory optimizations and their universal effect on different types of layers and various complete networks.

And finally, we also study the instruction cache efficiency in GPU architecture. As concurrent kernel execution becomes common for GPU computing, the large dynamic instruction footprint can make the small capacity of L1 Instruction cache a performance bottleneck. This study looks into the efficiency of instruction cache on both single kernel and concurrent kernel executions. For the workloads with high instruction cache contention, we employ efficient memory prefetching mechanisms to improve their performance for both single-kernel execution and concurrent-kernel executions.
REFERENCES


