

Locality Enhancement and Dynamic Optimizations on Multi-Core and GPU

Zheng Zhang

Taixing, Jiangsu, China

Master of Science, The College of William and Mary, 2007

Bachelor of Science, Shanghai Jiao Tong University, 2004

A Dissertation presented to the Graduate Faculty
of the College of William and Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

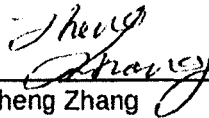
The College of William and Mary
August 2012

Copyright © Zheng Zhang
All Rights Reserved

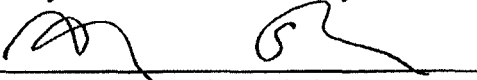
APPROVAL PAGE

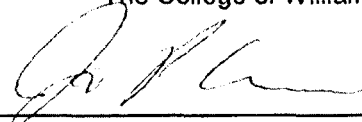
This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

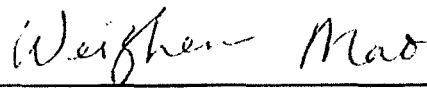
Doctor of Philosophy

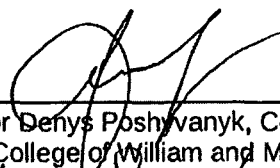

Zheng Zhang

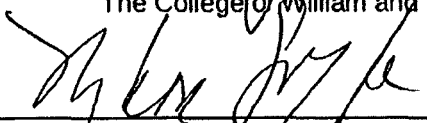
Approved by the Committee, May 2012


Committee Chair
Assistant Professor Xipeng Shen, Computer Science
The College of William and Mary


Associate Professor Phil Kearns, Computer Science
The College of William and Mary


Associate Professor Weizhen Mao, Computer Science
The College of William and Mary


Assistant Professor Denys Poshyvanyk, Computer Science
The College of William and Mary


Professor Mary Lou Soffa, Computer Science
The University of Virginia

ABSTRACT PAGE

Enhancing the match between software executions and hardware features is key to computing efficiency. The match is a continuously evolving and challenging problem. This dissertation focuses on the development of programming system support for exploiting two key features of modern hardware development: the massive parallelism of emerging computational accelerators such as Graphic Processing Units (GPU), and the non-uniformity of cache sharing in modern multicore processors. They are respectively driven by the important role of accelerators in today's general-purpose computing and the ultimate importance of memory performance. This dissertation particularly concentrates on optimizing control flows and memory references, at both compilation and execution time, to tap into the full potential of pure software solutions in taking advantage of the two key hardware features.

Conditional branches cause divergences in program control flows, which may result in serious performance degradation on massively data-parallel GPU architectures with Single Instruction Multiple Data (SIMD) parallelism. On such an architecture, control divergence may force computing units to stay idle for a substantial time, throttling system throughput by orders of magnitude. This dissertation provides an extensive exploration of the solution to this problem and presents program level transformations based upon two fundamental techniques – thread relocation and data relocation. These two optimizations provide fundamental support for swapping jobs among threads so that the control flow paths of threads converge within every SIMD thread group.

In memory performance, this dissertation concentrates on two aspects: the influence of non-uniform sharing on multithreading applications, and the optimization of irregular memory references on GPUs. In shared cache multicore chips, interactions among threads are complicated due to the interplay of cache contention and synergistic prefetching. This dissertation presents the first systematic study on the influence of non-uniform shared cache on contemporary parallel programs, reveals the mismatch between the software development and underlying cache sharing hierarchies, and further demonstrates it by proposing and applying cache-sharing-aware data transformations that bring significant performance improvement. For the second aspect, the efficiency of GPU accelerators is sensitive to irregular memory references, which refer to the memory references whose access patterns remain unknown until execution time (e.g., $A[P[i]]$). The root causes of the irregular memory reference problem are similar to that of the control flow problem, while in a more general and complex form. I developed a framework, named G-Streamline, as a unified software solution to dynamic irregularities in GPU computing. It treats both types of irregularities at the same time in a holistic fashion, maximizing the whole-program performance by resolving conflicts among optimizations.

Table of Contents

Dedication	vi
Acknowledgements	vii
List of Tables	viii
List of Figures	x
1 Introduction	1
1.1 Performance Impact of Divergent Control Flows	2
1.2 Handling Irregular Memory Reference Patterns	3
1.3 Matching Non-Uniform Cache Sharing Hierarchy	4
1.4 Dissertation Contributions	5
1.5 Dissertation Layout	6
2 Divergent Controls Flows	8
2.1 Motivation	8
2.2 GPU Architecture and CUDA Programming Model	11
2.3 CUDA Kernels and Thread-Data Remapping	13
2.3.1 An Abstract Form of GPU Kernels	13
2.3.2 Concept of Thread-Data Remapping	14

2.4	Mechanisms to Realize Thread-Data Remapping	16
2.4.1	Reference Redirection	16
2.4.2	Data Layout Transformation	17
2.4.3	Mechanism Selection	18
2.5	Data Layout Transformation On the Fly	19
2.5.1	Hiding Overhead through CPU-GPU Pipelining	20
2.5.2	Overhead Minimization through LAM	22
2.6	Evaluation	25
2.6.1	Methodology	26
2.6.2	3D-LBM	28
2.6.3	GAFORT	29
2.6.4	MarchingCubes	30
2.6.5	Reduction	32
2.6.6	BlackScholes	33
2.6.7	Summary of Experimental Results	34
2.7	Related Work	35
2.8	Summary	35
3	A Unified Framework for Both Control and Memory Irregularities	37
3.1	Motivation	37
3.2	Problem Definition	43
3.3	A Framework of Program Level Transformations	44
3.3.1	Two Basic Transformation Mechanisms	44
3.3.2	Three Transformation Methods	45
3.3.2.1	Data Reordering	45
3.3.2.2	Job Swapping	46
3.3.2.3	Hybrid Transformations	47
3.3.2.4	Comparisons	48

3.4	Determination of Desirable Data Layouts and Mappings	49
3.4.1	Irregular Memory References	49
3.4.1.1	Data Reordering	50
3.4.1.2	Job Swapping	52
3.4.2	Divergent Control Flows	53
3.4.2.1	Reference Redirection	53
3.4.2.2	Hybrid Approach	54
3.4.3	Co-Existence of Irregularities	54
3.5	Adaptive Efficiency Control	55
3.5.1	Dependence and Kernel Splitting	55
3.5.2	Approximation and Overlapping	56
3.5.3	Adaptive Control	57
3.6	G-Streamline Usage and Other Issues	61
3.6.1	Overview	61
3.6.2	Improved Sorting Efficiency	62
3.6.3	CPU Thread Affinity	62
3.7	Evaluation	63
3.7.1	Results Overview	65
3.7.2	Programs with Independent Loops	67
3.7.3	Programs with Loop-Carried Dependences	69
3.7.4	Program with No Central Loop	70
3.8	Related Work	70
3.9	Summary	72
4	Non-Uniform Cache Sharing Hierarchies	73
4.1	Overview	73
4.2	Experiment Design	76
4.2.1	Benchmarks	76

4.2.2	Factors	78
4.2.3	Measurement Schemes	80
4.3	Measurement and Findings	80
4.3.1	Sharing Versus Non-Sharing	81
4.3.2	Comparisons Among Sharing Cases	85
4.3.3	Pipeline Programs	87
4.3.4	Effects of Thread Binding	90
4.4	Program-Level Transformation	93
4.4.1	Streamcluster	93
4.4.1.1	Transformation	93
4.4.1.2	Performance	96
4.4.2	Blackscholes	99
4.4.3	Bodytrack	100
4.4.4	Ferret	101
4.5	Related Work	102
4.6	Summary	104
5	Cross-Input Adaptive Performance Tuning for GPUs	105
5.1	Motivation	105
5.2	Experiment Platform	107
5.3	Challenges for Optimizing GPU Programs	109
5.4	Adaptive Optimization Framework	110
5.4.1	Overview	111
5.4.2	Stage 1: Heuristic-Based Empirical Search and Data Collection	112
5.4.2.1	Optimization Pragmas and G-ADAPT Compiler	112
5.4.2.2	Performance Calibrator and Optimization Agent	114
5.4.3	Stage 2: Pattern Recognition and Cross-Input Adaptation	115
5.5	Evaluation	116

5.5.1	Matrix-Vector Multiplication	117
5.5.2	Parallel Reduction	119
5.5.3	Matrix Transpose	120
5.5.4	Other Benchmarks and Overall Results	120
5.6	Discussions	122
5.7	Related Work	123
5.8	Conclusion	124
6	Conclusion	125
	References	128
	VITA	138

*To my dear family,
for their unsurpassed love and continuous support.*

ACKNOWLEDGEMENTS

The research in this dissertation would not have been possible without the support of many people. First of all, I would like to thank my advisor Xipeng Shen for his selfless and persistent guidance throughout my time in graduate school. His kindness, diligence, and ability to think on his feet has made working with him a real pleasure.

I would like to thank my remaining committee members for their service and mentorship - Phil Kearns, Denys Poshyvanyk, Weizhen Mao, and Mary Lou Soffa. I thank Dr. Phil Kearns for his wise advice and especially for his sturdy support when I was facing difficulties in changing my research area in the first few years of my PhD study. I thank Dr. Poshyvanyk for being both my committee member and a great friend. I thank Dr. Weizhen Mao for her thoughtful comments and feedback on the theoretical aspect of my work. I have known Dr. Mary Lou Soffa on many occasions. She has been successful both as a researcher and a woman. She has inspired me in all different ways. I am very fortunate to have her on my committee as an external committee member and have her valuable feedback on my dissertation work.

I am grateful to my colleagues from the CAPS group for their collaboration - Yunlian Jiang, Ziyu Guo, Kai Tian, Zhijia Zhao, Mingzhou Zhou, Yufei Ding, Bo Wu and Fen Mao. We have made a great team together. I am also very grateful to the departmental staff Vanessa Godwin and Jacquelyn Johnson for their devoted administrative support. I benefited from a good work-life balance in graduate school because of all my kind and funny friends, in particular - Liuming Liu, Zhen Ren, Xin Ruan, Qian Si, Yufei Ding, Han Li, Bo Dong, Ziyu Guo, Mingzhou Zhou and Zhang Xu.

Lastly, I would like to thank my family for their unconditional support over all these years. I can't express how lucky I am to have them in my life. My parents, my husband and my younger brother, have always been there for me and supporting every major decision I have made in my life. They are great people themselves - loving, understanding, hardworking, kind, upbeat and honest. They are shining examples to me and I can never learn enough from them.

List of Tables

2.1	Benchmarks for divergence removal experiments	27
2.2	Experimental results of 3D-LBM	29
2.3	Experimental results of GAFORT	31
2.4	Experiment results of MarchingCubes	31
2.5	Experiment results of Reduction	32
3.1	Benchmarks for G-Streamline and dynamically determined optimization parameters	64
3.2	Numbers of thread divergences and memory transactions on one GPU SM reported by hardware performance counters	65
3.3	Breakdown of time for different memory transaction sizes	67
4.1	Benchmarks for non-uniform shared cache experiments	77
4.2	Dimensions covered in the measurement for shared cache experiments	78
4.3	Configuration of NUCA CMP machines	80
4.4	Maximal percentage of the performance differences caused by different bindings of threads to a given set of cores	87
4.5	Performance of <i>ferret</i> on the Intel machine with different thread placement on cores. (S: pipeline stage)	89
4.6	Streamcluster with a different input	97

5.1	Benchmarks for input adaptivity experiments on GPU	117
-----	--	-----

List of Figures

2.1	A piece of code adapted from <i>gafort</i> , a program implementing a genetic algorithm. Both the “if” statement and the “for” loop may cause thread divergence. (<i>tid</i> is the sequential number of the current thread.)	9
2.2	An abstract form of a GPU kernel containing conditional statements.	14
2.3	Illustration of thread-data remapping for elimination of thread divergences. The two types of circles represent two types of input data sets. Their differences make threads in a warp diverge on a condition statement. (For ease of illustration, warp size of 4 is used.) After the remapping, all the divergences disappear.	15
2.4	An example illustrating the (simplified) use of CPU-GPU pipelining to hide the overhead in thread-data remapping transformations. The code in the bottom box is part of the G-Streamline library.	20
2.5	The approximation scheme, LAM (label-assign-move), for reducing the overhead in data layout transformation.	26
2.6	Speedup with and without efficiency control.	34

3.1	Examples of dynamic irregularities (warp size=4; segment size=4). Graph (a) shows that inferior mappings between threads and data locations cause more memory transactions than necessary; graph (b) shows that inferior mappings between threads and data values cause threads in the same warp diverge on the condition.	38
3.2	Potential performance improvement when dynamic irregularities are eliminated for applications running on an GPU (Tesla 1060).	39
3.3	Major components of G-Streamline.	41
3.4	Examples for illustrating the uses of data reordering and job swapping for irregularity removal.	44
3.5	Using data relocation for job swapping faces some complexities.	47
3.6	Kernel splitting makes CPU-GPU pipelining remain feasible despite loop-carried dependences.	56
3.7	Dynamic adjustment for optimization ratio.	61
3.8	Speedup from thread-data remapping.	65
4.1	The running time of each program in the sharing case normalized to its running time in the non-sharing case. The bars in a group from left to right correspond to the cases of 2 threads on the <i>simlarge</i> input, 2 threads on the <i>native</i> input, 4 threads on the <i>simlarge</i> input, and 4 threads on the <i>native</i> input.	82
4.2	Comparisons of L2-cache accesses and misses for 2-thread cases on the Intel machine. (“S” for cache-sharing cases; “NS” for non-cache-sharing cases; “CM” for cache misses; “CA” for cache accesses.)	83
4.3	Read sharing for 2-thread cases on Intel. It is computed as the number of read accesses to the L2-cache lines with a “shared” state, normalized by the total number of L2-cache accesses.	84

4.4	Temporal traces of the L2 cache miss rates on the Intel machine when 4 threads are placed on the same set of cores differently.	88
4.5	Box plot of L2-cache miss rates per thread on the Intel machine when different thread-core assignments are used. (c1,c2,c3 refer to the top 3 configurations in Table 4.5.)	90
4.6	Effects of thread-core binding on Intel and AMD machines. Every box shows <i>min</i> , <i>max</i> and <i>median</i> (·) of the five runs for every configuration. The Y-axis is the running time in the non-binding case normalized to the average time in the binding case.	91
4.7	Simplified pseudo-code illustrating the original and optimized versions of the function <i>pgain()</i> in <i>streamcluster</i> . It is assumed that two threads constitute a sibling group that share cache.	94
4.8	Speedup by inter-thread, intra-thread, and combined transformations on the Intel machine. Adjacent two threads share L2 cache.	97
4.9	The reduction of L2 cache miss rates and memory bus contention on the Intel machine. In each bar group, the first two are when inter-thread transformation is applied to the original program, and the next two are when the combined transformation for both intra-thread and inter-thread sharing is applied. In each pair, the two bars respectively correspond to the cases when the sibling threads of the inter-thread optimized version run on two cores sharing or not sharing cache.	98
4.10	The reduction of L2 cache misses of <i>blackscholes</i> , <i>bodytrack</i> and <i>ferret</i> due to cache-sharing-aware transformation. The Intel machine is used.	100
5.1	G-ADAPT: An adaptive optimization framework for GPU programs.	111
5.2	Performance of matrix-vector multiplication on two inputs when different optimization decisions are used. LU: loop unrolling levels; THRD: number of threads per block. The maximum unrolling level can only be $\sqrt{THRD}/4$	118

5.3	The best values of the optimization parameters of <i>mvMul</i> are input-sensitive. G-ADAPT addresses the influence and produces significant speedup compared to the original program.	119
5.4	Experimental results on <i>reduction</i>	119
5.5	Experimental results on <i>transpose</i>	120
5.6	The ranges of speedup brought by different optimization decisions. For each program, the left bar shows the range of speedup (less than 1 means slow-down) if the worst decision is taken. The right bar shows the range of speedup when the G-ADAPT's prediction is used.	121

Chapter 1

Introduction

Today's parallel computing systems are imposing multi-fold challenges on programmers by getting more diversified and complicated. Modern parallel architectures are equipped with unprecedented parallelism and heterogeneity. Examples are GPGPUs (General Purpose Graphic Processing Units) and Intel's MIC (Many Integrated Core) chips. Programmers not only have to handle the complexities in choosing and learning different parallel programming languages, but also generate efficient code to harness the tremendous hardware computing power. Not to mention that program inputs and execution environments make the optimization parameters dynamic and unpredictable. Many existing algorithms, program analysis and development tools are designed for sequential programs or outdated parallel programming models. In this dissertation, I have examined the existing parallel programming paradigms, identified the major challenges for producing efficient parallel code, and developed programming system support for tackling these challenges.

They are three main performance challenges: divergent control flow problem that causes under-utilization of processor cores for many-core GPGPUs (General Purpose Graphic Processing Units) with massive parallelism, irregular memory reference that increases unnecessary memory accesses for SIMD architectures, and non-uniform cache hierarchies that may create not only synergistic sharing but also resource contention, which is detrimental to program efficiency on commodity multi-core processors. For the first two challenges,

this dissertation provides an extensive exploration of the causes and solutions, including enhanced understanding for relations between thread and dataset mappings, program level transformations based on two fundamental techniques – thread relocation and data relocation, a unified framework for applying the program transformations to address different challenges, the theoretical analysis for optimality of the transformations, as well as runtime system that provides adaptive efficiency control to manage the transformation overhead and make the transformation adapt to different program inputs and phases. For the third challenge, this dissertation presents the first systematic study on the influence of non-uniform shared cache on contemporary parallel programs, reveals the mismatch between the software development and underlying cache sharing hierarchies, and further demonstrates it by yielding significant speedup through novel cache sharing aware transformations. Sections 1.1, 1.2, 1.3 describe the three major challenges in details and the overview of solutions proposed in this work.

1.1 Performance Impact of Divergent Control Flows

Conditional branches cause divergences in program control flows, which may result in serious performance issues on massively parallel architectures equipped with Single Instruction Multiple Data (SIMD) parallelism. On such an architecture, control divergence may force many computing units to stay idle for a substantial time, throttling system throughput by orders of magnitude. My research has addressed this problem by proposing two types of program transformations for applications on a popular SIMD like architecture Graphic Processing Units (GPUs) [74]. The first optimization is thread relocation, which reorganizes logical threads by reassigning their thread IDs to minimize the divergences among their control flows. The second is data relocation, which switches the tasks of threads by relocating data on memory without changing thread IDs and data reference indices. These two optimizations are fundamental techniques for swapping jobs among threads so that the control flow paths of threads converge within every SIMD group. Compared to previously

proposed hardware extensions, the software approaches are readily deployable and address dynamic program behavior by adapting to different program inputs and phases. This work is the first in a series of research efforts that addresses control flow divergence problem on GPUs in software. Experiment results have demonstrated that these transformation techniques reduce the amount of thread divergence significantly in many applications.

1.2 Handling Irregular Memory Reference Patterns

The efficiency of massively data parallel architectures such as GPGPUs is sensitive to dynamic irregular memory references, which refer to the memory references whose access patterns remain unknown until execution time (e.g., $A[P[i]]$). Experiments have shown great performance gains when these irregularities are removed. But it remains an open question how to achieve those gains through software approaches. I present in this dissertation a systematic exploration on this problem [76]. I discovered that the properties and solutions to the irregular memory reference problem share similarities with that of the control flow problem.

I explored the inherent properties of both of them, including their interactions, their relations with program data and threads, the computational complexities in removing them, and heuristics-based algorithms for their removal through data reordering, job swapping, and hybrid transformations. Based on these findings, I developed a framework, named G-Streamline, as a unified software solution to dynamic irregularities in GPU computing. It treats both types of irregularities at the same time in a holistic fashion, maximizing the whole-program performance by resolving conflicts among optimizations. Its optimization overhead is largely transparent to GPU kernel executions, preserving at least the efficiency of the original un-transformed GPU application. More importantly, it is robust to the presence of various complexities in GPU applications. Our experiments results demonstrate that G-Streamline is effective in reducing both types of dynamic irregularities and is capable of producing significant performance improvements for a variety of applications.

1.3 Matching Non-Uniform Cache Sharing Hierarchy

As the number of cores increases in a single chip, the sharing of the memory hierarchy becomes deeper, more heterogeneous and more complex. This causes cache contention, increases conflicts and meanwhile can also increase synergistic sharing. A comprehensive understanding on the implication of this sharing behavior is needed. Many studies concentrate on the interaction of co-running programs and the solutions are mostly in the areas of architecture and operation systems. In my work presented in [75], I conducted the first extensive investigation of the influence of shared cache on contemporary multi-threaded applications, with many important factors systematically examined, including threads scheduling schemes, parallelization model, architecture and so on. For threads in a multithreaded application, our study shows that whether a cache is shared exhibits surprisingly minor effects on the performance of the application. After identifying the reasons, I develop shared-cache-aware program transformations, which produces asymmetric relations among threads to match the non-uniform cache sharing in modern multi-core processors. I further enhanced the depth of the study through an empirical examination of the relations between the intra-thread sharing (private cache reuse) and inter-thread sharing (shared cache reuse) using more case studies with different memory behaviors [77].

Other work One important trend in computing is the development of heterogeneous systems, exemplified by the integrations of general-purpose cores with special-purpose processors, including GPU, FPGA, DSP. The heterogeneity in modern computing systems brings new opportunities but also the new challenges for application development and optimization. In my work [76], [72], I explored the creation of a whole system synergy by making multicore CPU and GPU work cooperatively.

I proposed a pipeline scheme, which assembles different types of processors together into a pipeline so that one can optimize for the computations of another in an asynchronous and transparent manner. I developed kernel splitting to circumvent the issues caused by data dependences, and created an adaptive controller to help tailor the configurations of the

pipeline on the fly to best suit the need of the current execution. Experiments show that the pipeline scheme provides an effective way to maximize the throughput of a heterogeneous system.

Another urgent issue in modern parallel computing is the adaptivity to runtime environments and program inputs. As mentioned earlier, the optimization parameters vary when the program input and environment changes (like architecture and co-running processes) and the optimization space explodes because of the large number of parameters including both configuration parameters (like number of threads) and program parameters (like loop unrolling level). I developed a framework called G-adapt that helps train and predict best optimization parameter according to program inputs by machine learning techniques.

1.4 Dissertation Contributions

First of all, this dissertation presents an unified overview of the two types of problems in GPU computing - the control divergence and irregular memory reference. These two problems are similar in that they are both caused by the differences among threads in the same SIMD thread group. The different executions paths caused by control flow statements may lead to severe underutilization of cores within every SIMD processor. The differences in the memory segment which every thread's data access fall into within a SIMD thread group lead to more memory transactions than necessary. The relations among different threads either in control or memory pattern are extremely important in today's parallel system with respect to computing efficiency. In this dissertation, I propose a program level transformation framework to address these two types of seemingly different problems based on two fundamental techniques - thread relocation and data relocation. Thread relocation and data relocation can be further combined to obtain the benefits and mitigate the side effects of both transformations. The second contribution made by this dissertation is the exploration of the influence of non-uniform cache on modern parallel program development. The results of my study indicate that there is great potential in exploiting the non-uniform

synergistic cache sharing through program transformations that combine thread scheduling and data reordering. Likewise, the implication is that the relations among threads that run on processor cores sharing cache are extremely important. The difference from the first contribution is that the thread group here are the threads that share cache, while the thread group mentioned above are threads that run on every SIMD processor.

The third contribution made by this dissertation is enhanced parallel program adaptivity to architecture, runtime environment and application inputs. I designed and implemented a CPU/GPU pipelining runtime system that dynamically monitors and controls program transformation overhead through asynchronous CPU/GPU task scheduling, termination and coordination. This CPU/GPU pipelining runtime system ensures that the transformation overhead is transparent and that the transformation level can be adjusted to strike a balance between its overhead and benefits. Additionally, I developed an input-adaptive performance tuning framework for GPGPU programs with my colleague Yixun Liu. It learns the pattern between program inputs and best program transformation parameters through machine-learning techniques, and provide the optimally transformed program given a new input.

1.5 Dissertation Layout

The layout of this dissertation is as follows. Chapter 2 focuses on the first challenge - the control divergence in GPU programs, and elaborates my explorations in removing control divergence through software techniques. Chapter 3 focuses on the second challenge - the irregular memory reference problems. It first reveals the connections between the control divergence and memory reference problems - the same reason of inappropriate thread-data mapping. I refer to these two types of problems as irregularities in GPU programs, and proposed a unified framework of solutions based on two fundamental program level transformations: thread relocation and data relocation. Chapter 4 discusses implication of non-uniform cache sharing and demonstrates the potential of cache sharing aware program

transformations. Chapter 5 presents my work on cross-input adaptive performance tuning for GPU programs.

Divergent Controls Flows

2.1 Motivation

Because of their remarkable computing power and cost efficiency, GPUs (Graphics Processing Units) have emerged as a kind of influential platform for high performance computing [6, 19, 45, 50, 51, 56].

However, as GPUs are specially designed for massive data-parallel computing, their performance is subject to the presence of condition statements in a GPU application. When a GPU application runs, a group of threads (called a *thread warp*¹) are deployed in each GPU SM (streaming multiprocessor) so that they can run concurrently to maximize the usage of the computing power. Normally, the threads in a warp run in a SIMD (Single Instruction Multiple Data) fashion. However, on a conditional branch where the threads diverge in which path to take, the threads taking different paths have to run serially. This phenomenon is called *thread divergence*.

Thread divergence often causes serious performance degradations. Figure 2.1 shows a piece of code adapted from a program named *gafort* that performing a genetic algorithm. Suppose the first 32 elements in *r1* are all even numbers except *r1*[13]. When the first warp encounters the “if” statement in the code, only thread 13 passes through the check and

¹This paper uses NVIDIA CUDA terminology and programming model for discussions. A warp is assumed to contain 32 GPU threads.

```

if (r1[tid]%2){
    .....// compute1
}
.....
icross = sqrt (r2[tid]);
for (n= icross; n < nchrome; n++){
    .....// compute2
}

```

Figure 2.1: A piece of code adapted from *gafort*, a program implementing a genetic algorithm. Both the “if” statement and the “for” loop may cause thread divergence. (*tid* is the sequential number of the current thread.)

conducts *compute1*, while all the other 31 threads are idle and waiting. Note that because the warp is not completely idle, no other warps are allowed to run on that SM during that time. So at most 1/32 computing power of the SM is being used. The similar problem exists on the “for” loop in Figure 2.1. Suppose that the first 32 elements in *r2* are all as large as *nchrome* except that *r2*[4] is 0; then all the 31 threads have to stay idle and wait until thread 4 finishes its *nchrome* (which could be very large) iterations of *compute2*, causing substantial waste of computing power. In reality, we observe up to 1.47 speedup when thread divergences are removed (as shown in Section 5.5), which echos the potential observed in previous studies [11, 24].

As a side effect of the architectural support of GPUs for massive data parallelism, this divergence problem exists in virtually all types and generations of modern GPUs. It impairs the adoption of GPU for many applications that contain non-trivial condition statements. There have been limited solutions proposed, among which, some [11] aim at reducing register pressure incurred by thread divergences rather than the divergences themselves, some [24] tackle divergences directly but rely on special hardware support.

In this chapter, we propose a pure software solution via runtime thread-data remapping. The basic scheme is simple: to switch the data sets that the GPU threads work on so that all the threads in a warp would take the same path on a conditional branch. Consider the “if” statement in the example mentioned earlier in Figure 2.1. Suppose the other (say 992) elements of *r1* all have the similar value pattern as the first 32 elements have—that is, in every 32 elements, only one value is odd—we can remove all the thread divergences by

remapping the threads to data so that the first 992 threads all work on the data with small $r1$ values, and only the final 32 threads work on the large ones. This strategy, apparently, works with the “for” loop as well in a similar manner.

A series of issues must be solved before thread-data remapping can be feasibly applied to real GPU applications. The first is the determination of a desirable thread-data mapping. In a real GPU application, data accesses may be irregular or have complex indexing expressions. The thread-data remapping may cause side effects—such as, altering original regular memory reference patterns—and hurt other performance factors (e.g., memory accesses as detailed in Section 2.4).

The second issue is on what mechanism to use to realize the thread-data remapping. There are two options. One is through *reference redirection* (also called indirect accessing). For the “if” example in Figure 2.1, we may create an index array $I[]$ and change $r1[tid]$ to $r1[I[tid]]$. Appropriately setting the values in $I[]$ will produce a desired thread-data mapping. The second option is through *data layout transformation* (also called data packing). For the “if” example again, if we keep the kernel unchanged but relocate the elements in $r1$ so that small values are all at the front of $r1$ and large ones all at the end, we can achieve the same mapping as the redirection array produces. It is necessary to explore both options to determine their limitations, effectiveness, and how they should be implemented and applied safely.

The final but also the most difficult issue is on the conflict between the large remapping overhead and the need for the remapping to happen on the fly. Because in most cases the values of the data set that a condition statement depends on are not known until run time, the thread-data remapping must happen on the fly. However, the remapping, through either redirection or layout transformation, typically causes significant overhead that may easily outweigh the remapping benefits. It is hence crucial to minimize or hide the overhead, as well as to protect the basic efficiency of the GPU application from getting jeopardized by the remapping process.

In this work, we develop a set of techniques to address these issues, making run-time

thread-data remapping feasible for GPU computing. Our description starts with an abstract form of GPU applications and the concept of thread-data remapping (Section 2.3). In Section 2.4, we discuss the two mechanisms, reference redirection and data layout transformation, for the realization of thread-data remapping, with their properties, constraints, and suitable scenarios uncovered.

In Section 3.5, we present two techniques that make run-time remapping possible for GPU computing by effectively hiding and reducing remapping overhead. The first technique is a CPU-GPU pipelining scheme, which allows the remapping-related operations to overlap with GPU kernels execution. Its effectiveness in hiding remapping overhead comes from its exploitation of the massive data-parallelism in GPU applications and the independence between CPU and GPU memory systems. The second technique is a linear-time LAM (label-assign-move) scheme, which minimizes the number of data movements required for the generation of a target thread-data mapping. Together, the two techniques hide virtually all remapping overhead for most applications and turn costly remapping affordable for runtime deployment.

Section 5.5 reports up to 1.47 speedup on a set of GPU applications, demonstrating the effectiveness of the techniques for eliminating thread divergences and streamlining GPU computing on the fly.

2.2 GPU Architecture and CUDA Programming Model

A GPU chip consists of several multi-core processor, each of which is a SIMD (Single Instruction Multiple Data) engine. In NVIDIA terminology, it is referred to as Streaming Multiprocessor (SM). For the rest of the dissertation, we use NVIDIA terminology to explain the concepts and describe the techniques. Every streaming multiprocessor typically consists of 8, 16, or 32 cores. There is one instruction issuing unit per SM. The implication is that every core has to execute the same instruction at the same time. Different SMs can execute different instructions at one time. There are two types of memory on a GPU: the on-chip

memory and the off-chip memory. Based on the types of usages, there are shared memory, constant memory, texture memory, and etc. On-chip memory can be accessed much faster than off-chip memory, but has small capacity. *Shared memory* is a type of on-chip memory that works like scratch pad memory. Users have to manually read, write and maintain the consistency. *Constant memory* is read-only off-chip memory that can be automatically cached by hardware. *Texture memory* is also off-chip memory, the data in which can be cached by hardware. Besides, it has two-dimensional locality, which means data adjacent in both rows and columns can be accessed together very fast. Off-chip memory has larger capacity, but larger access. It is referred to as *global memory*. GPUs also have caches. There is a read-coherent private L1 cache for every SM. All NVIDIA general purpose GPU cards, from Fermi architecture and later, have a fully coherent L2 cache shared by all SMs.

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model, which includes language extensions to C/C++, a runtime and driver level API access to NVIDIA GPUs. Besides, CUDA has a mature ecosystem with supporting libraries and development tools provided and used by a large community of CUDA programmers. The work in this dissertation is based on CUDA since it is efficient and meanwhile the leading GPU programming mode for general purpose GPUs. Writing a CUDA program is like writing a C/C++ program. With CUDA language extensions, users specify which functions should run on GPUs. These functions are referred to as *kernel* functions. Every *kernel* function is executed by many GPU threads. These threads that execute the same kernel are organized into a *grid*. A *grid* of threads are further divided into many blocks. Threads within a block can communicate using *shared memory* and threads across blocks can only communicate using *global memory*. Threads within a block can also be synchronized using hardware support. Threads across blocks can be implemented using software approaches. The number of threads to execute the function is much larger than the total number of processing units and therefore they run simultaneously. A GPU card has very large register file and they are partitioned into register banks dedicated for every thread. The *shared memory*, if used, is partitioned to serve every thread as well. When a

grid of threads are launched, the runtime starts allocating resources such as registers and shared memory for as many active threads as possible. The active threads will start running on the SMs until they have reached the end of the kernel function and another batch of threads will be loaded in. The number of active threads needs to be larger than or equal to the number of processor cores to keep all of them utilized. A block of threads are further partitioned into warps. A warp is the minimal execution and scheduling unit. When one warp of threads are waiting for memory accesses to finish, another warp can be switched in to fill the instruction pipeline, which can be analogical to very *lightweight context switch*.

2.3 CUDA Kernels and Thread-Data Remapping

This section first outlines an abstract form for GPU kernels that contain condition statements, based on which, it introduces the concept of thread-data remapping.

2.3.1 An Abstract Form of GPU Kernels

As mentioned in Section 2.2, GPU application written in CUDA consists of some CPU code and GPU code. The GPU part includes one or more functions; each of them is called a *GPU kernel*. All applications start from the “main” function in the CPU code. When the CPU launches a GPU kernel, a number of GPU threads are created with each having a unique sequential number, called thread ID and denoted as *tid* in this paper² Upon its creation, every thread starts an instance of the GPU kernel independently. Although they execute the same kernel with the same parameter values, they may behave differently and access different data. The differences usually stem from the uses of *tid* inside the GPU kernel.

Figure 2.2 outlines an abstract form for GPU kernels containing condition statements. We omit the parts irrelevant to thread divergence. We use arrays to represent container objects in GPU kernels as they are the most commonly used data structures. The *input arrays* are those arrays whose values are passed from CPU to GPU at the invocation of the

²More precisely, CUDA uses several builtin variables to store the index of the current thread block and the index of the current thread in its block. Combined together, they form the *tid*, a unique identity for the current thread.

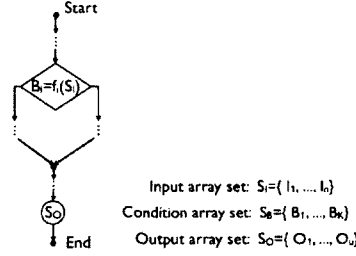


Figure 2.2: An abstract form of a GPU kernel containing conditional statements.

kernel. Note, in this abstract form, all thread IDs together are viewed as a special input array as $IDArray[tid] = tid$. This view is important for the applicability of thread-data remapping (as Section 2.4.2 will show).

Each *condition array* corresponds to one instance of a condition statement in the kernel, storing only binary values: $B_i[tid] = 1$ means that thread tid goes through the check of the i th condition; $B_i[tid] = 0$ means otherwise. A loop or a condition with more than two branches are viewed as a series of such binary conditions. The denotation $f_i()$ represents the computation that produces B_i from the input arrays. The *output arrays* store the ultimate computation results.

Additionally, we introduce the concept of path vectors. A *path vector* of a thread is

$$S_B[tid] = \langle B_1[tid], B_2[tid], \dots, B_K[tid] \rangle .$$

It summarizes the entire path taken by thread tid . It is easy to see that a warp diverges if and only if there exist two threads in the warp whose path vectors differ. Another important concept is the *input set* of a thread. It refers to the set of elements in all the input arrays that are accessed by a thread, denoted as $S_I[tid]$.

2.3.2 Concept of Thread-Data Remapping

It is important to note that the input set of a thread completely decides how that thread behaves in a given kernel, hence deciding the path vector of that thread. The implication

Figure 2.3: Illustration of thread-data remapping for elimination of thread divergences. The two types of circles represent two types of input data sets. Their differences make threads in a warp diverge on a condition statement. (For ease of illustration, warp size of 4 is used.) After the remapping, all the divergences disappear.

is that, if after a remapping, the i th thread maps to the original $S_I[j]$ ($i \neq j$), then the new value of $S_B[i]$ would be the same as the original value of $S_B[j]$. This is the basic rationale for using thread-data remapping to remove thread divergences. If we view the values of the path vector produced by an input set as the color of that input set, there is no divergence if and only if all threads in a warp are mapped to the input sets of the same color. So the thread-data remapping in this work is essentially to find an appropriate mapping between threads and input sets, as Figure 2.3 illustrates.

Thread divergences may come from two kinds of sources. The first is the differences in loop trip-counts (i.e. numbers of iterations), as illustrated by the “for” loop in Figure 2.1. Clearly, the time for a thread warp to finish the execution of the loop is determined by the largest trip-counts of the loop in the executions by all the threads in that warp. Each iteration of a loop typically takes a similar amount of time to run. Hence, the total time that the loop costs all the warps is determined by

$$T = \sum_{i=1}^W \max_{k \in \text{warp}_i} (it_k),$$

where W is the total number of warps, it_k is the trip-count of the loop executed by the k th thread. Minimizing thread divergences hence leads to the minimization of T . Our basic strategy is to create a thread-data mapping so that the trip-counts of the loop executed by the threads form a sorted sequence—that is, after the remapping, $i < j \Rightarrow it_i \leq it_j$.

Thread divergence may also come from non-loop condition statements, as illustrated by the “if” example in Figure 2.1. In this case, the kernel contains no diverging loops but K ($K > 0$) other types of condition statements. Each thread hence has a K -dimensional path vector. From now on, we say two threads are of the same type if their path vectors are equal. The basic remapping strategy in this case is to greedily pack threads of the same

type together.

Although the remapping strategies in both cases are straightforward, realizing them safely, efficiently, and beneficially requires solutions to a series of issues.

2.4 Mechanisms to Realize Thread-Data Remapping

There are two ways to realize a thread-data remapping: reference redirection and data layout transformation. Although they have both been studied in many CPU program optimizations (e.g., [14, 17]), their applications in GPUs appear to face some new complexities. This section first describes the implementation and applicability of each of the two remapping approaches, and then discusses their respective strengths and weaknesses.

2.4.1 Reference Redirection

Reference redirection creates an index array, denoted as $D[\]$, to generate a new thread-data mapping. If a new mapping requires thread i map to the original input data set $S_I[j]$, then $D[i] = j$. This creation step is put into the CPU code before the invocation of the GPU kernel (actually in a pipelining fashion as detailed in Section 3.5). The index array D is passed to the GPU kernel at its invocation.

The transformation to the GPU kernel is simple. At the beginning of the kernel, a statement like “`__newtid = D[tid]`” is inserted, where `__newtid` is a new local variable³. The transformation then replaces all occurrences of `tid` in the kernel with `__newtid`.

The transformation is applicable to kernels that contain no dependences across threads. It can be proved that in this case, the transformed program produces the same output as the original program does. The intuition of the proof is simple: After the transformation, the computation and data accesses in all kernel instances remain the same as before, even though those kernel instances are executed by different threads. It is not hard to see that the transformation can also apply to the cases where a kernel consists of multiple sections

³Because CUDA uses several builtin variables for thread indexing, in the actual implementation, a new local variable is created for each of them, and their values are derived from the index array $D[\]$.

separated by barriers and inter-thread dependences exist only across sections but not within a section ⁴.

2.4.2 Data Layout Transformation

The second mechanism is data layout transformation. If the new mapping requires thread i map to the original input set $S_I[j]$, in this transformation, the content of $S_I[j]$ is copied to the corresponding locations in $S_I[i]$ before the invocation of the GPU kernel (again in a pipelining fashion, shown in Section 3.5). For instance, suppose in the original program, threads i, j, k map to $I_1[i]$, $I_1[j]$, and $I_1[k]$ respectively, but the new mapping requires these threads process $I_1[j]$, $I_1[k]$, $I_1[i]$ respectively. The transformation creates a new array I'_1 with $I'_1[i] = I_1[j]$, $I'_1[j] = I_1[k]$, and $I'_1[k] = I_1[i]$, and then replaces I_1 with I'_1 at the invocation of the GPU kernel. After the invocation of the GPU kernel, a restoration step is sometimes necessary, in which, the elements in the output arrays are reordered so that they are consistent with what the original program produces.

For the transformation to be safe, we require the GPU kernel meet two conditions:

1) The input sets of two arbitrary GPU threads have no overlap—that is, $S_I[i] \cap S_I[j] = \emptyset$ ($i \neq j; 0 < i, j < N$). And no two threads write to a common memory location.

2) For an arbitrary input data element accessed by thread i , there must be a counterpart in the input data set of thread j . Specifically, if $I_x[f(i)] \in S_I[i]$, where $f(i)$ represents the relation between the index of the data element and the thread ID, then $I_x[f(j)] \in S_I[j]$ ($0 < i, j < N$).

These two conditions ensure that the data movements cause no mistaken data overwriting. Fortunately, many GPU programs satisfy such conditions because of their data-level parallelism and simple dependences among threads.

Theoretically speaking, there is another condition to meet: The only effect of the thread

⁴In CUDA, since barriers work only for threads within a block (containing many warps), in the cases with barriers, the remapping transformation applies inside each thread block.

ID, tid , on the kernel execution is on deciding which data elements of the input and output arrays a thread references. The thread ID tid must not involve directly in value calculation. For example,

$$O_1[tid] = I_1[tid * 2] + I_2[tid]$$

is allowed, but the following one is not:

$$O_1[tid] = I_1[tid * 2] + tid$$

This condition is necessary for the correctness of the computation results. To see this point, one may consider the case where $I_1[2] = -1$ and $I_1[4] = 1$ for the example statement $O_1[tid] = I_1[tid * 2] + tid$. Suppose the remapping requires thread 1 map to $I_1[4]$ and thread 2 map to $I_1[2]$. After the data layout transformation, the execution of the example would produce $O_1[1] = 2$ and $O_1[2] = 1$, which differ from the original output $O_1[1] = 0$ and $O_1[2] = 3$. Apparently, the error cannot be corrected by the operations (i.e. reverse data movements) in the subsequent restoration step.

Fortunately, this third condition is easy to meet through a preprocessing step. In that step, an assistant array $IDArray$ is created before the GPU kernel invocation; its elements are set as $IDArray[tid] = tid$. The array is then passed to the GPU kernel at the kernel's invocation as an extra input array. Inside the kernel, all references to tid are replaced with $IDArray[tid]$. This transformation is demonstrated in the benchmark named *reduction* in Section 5.5.

For efficiency, we use asynchronous memory copy for data transfers between the host and GPU. In some kernels, the condition arrays determine the loop trip-counts but meanwhile are modified inside the loop body. This work does not handle such cases.

2.4.3 Mechanism Selection

The two mechanisms have their respective strengths and weaknesses. The applicability of data layout transformation is subject to some conditions as described in the previous section. In addition, the transformation usually causes larger transformation overhead that

the alternative because of the data movements and restoration it requires (although this problem can be alleviated as to be shown in Section 3.5).

On the other hand, data layout transformation maintains certain memory reference patterns of the original GPU kernel, whereas, reference redirection does not always do so. In GPU, memory reference patterns strongly affect the effective memory bandwidth. For instance, in NVIDIA G200, if the words accessed by a warp fall into n different segments of global memory (a segment contains 32, 64, or 128 consecutive bytes), the GPU needs to conduct n memory transactions for those accesses. (When the threads in a warp access memory locations in a small range, all the references by that warp may take only one transaction; such references are termed *coalesced memory references*.) So the changes that reference redirection causes to memory reference patterns may result in significant increases in the number of memory transactions for a kernel, and hence offset the benefits brought by the reduction of thread divergences.

In our implementation, the principle for the selection of these two mechanisms is as follows. If the reference redirection may be proved to hurt no memory reference efficiency, it is selected. Such cases may happen in two scenarios. One is that the kernel uses texture memory rather than global memory for data references: Texture memory tolerates memory pattern changes better than global memory for its use of cache. The other is that the redirection applies to calculations but not data references in the kernel. In other situations, the alternative mechanism, data layout transformation, is selected if it is applicable. Section 5.5 demonstrates the selection on different benchmarks.

2.5 Data Layout Transformation On the Fly

Because the values that a condition statement produces typically depend on the input data sets of the GPU application, thread-data remapping often needs to be applied during run time. However, the operations involved in the remapping are expensive. Without a careful design, the overhead may easily outweigh the benefits brought by the reduction

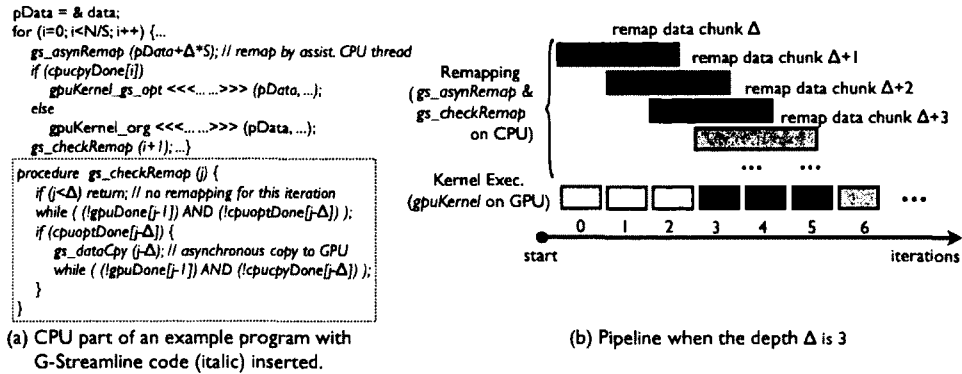


Figure 2.4: An example illustrating the (simplified) use of CPU-GPU pipelining to hide the overhead in thread-data remapping transformations. The code in the bottom box is part of the G-Streamline library.

of thread divergences. This section describes how we enable efficient runtime thread-data remapping through the use of CPU-GPU pipelining and a linear-time LAM scheme to hide and minimize remapping overhead.

2.5.1 Hiding Overhead through CPU-GPU Pipelining

The basic idea of the CPU-GPU pipelining is to make transformations happen asynchronously on CPU when GPU kernels are running. We first explain how the pipelining works in a setting where the main body of the original program is a loop. Each iteration of the loop invokes a GPU kernel to process one chunk of data; no dependences exist across loop iterations. This is a typical setting in real GPU applications that deal with a large amount of data.

Figure 2.4 shows an example use of the CPU-GPU pipelining. The CPU part of the original program is in normal font in Figure 2.4 (a). Each iteration of its central loop invokes *gpuKernel* to make the GPU process one chunk of the data. The italic-font lines are inserted code to enable the pipelined thread-data remapping. All functions with the prefix “gs-” are part of the G-Streamline library. Consider that the execution of the *i*th iteration of the loop. At the invocation of “*gs_asynRemap ()*”, the main CPU thread wakes up an assistant CPU thread. While the assistant thread does thread-data remapping for the chunk

of data that is going to be used in iteration $(i + \Delta)$, the main thread moves on to process the i th chunk of data. It first checks whether the G-Streamline transformation (started in the $(i - \Delta)$ th iteration) for the current iteration is already done. If so (i.e., *cpucpyDone*[i] is true), it invokes the optimized GPU kernel; otherwise, it uses the original kernel. While the GPU executes the kernel, the main CPU thread moves on to “gs_checkRemap ($i + 1$)” (pseudo code in the box) to copy the transformed $(i + 1)$ th chunk of data from host to GPU. This copying is conditional: The first while loop in “gs_checkRemap ()” ensures that the copying starts only if the transformation completes before the i th GPU kernel invocation finishes. The second “while” loop ensures that the main thread moves on normally without waiting for the data copying to finish if the i th GPU kernel invocation has completed. These two “while” loops together guarantee that the transformation and associated data copying cause no delay to the program execution even in the worst case.

The status arrays, *gpuDone*, *cpuoptDone*, and *cpucpyDone* in Figure 2.4, are conceptual. The “*cudaStreamQuery*” in CUDA API is actually used for checking the GPU kernel status.

It is worth noting that the presence of such loop structures is not mandatory for the pipelining scheme to work. Consider a program containing two phases of computation, whose second phase is a GPU kernel with thread divergences. If the divergences do not depend on the computation in the first phase, the remapping function for the second kernel may be invoked before the first phase so that the remapping can overlap with the first phase execution.

On the other hand, a common pattern in many GPU programs is that each thread works on a separate set of data. Inter-thread communications exist only within a block of threads; no global synchronizations are allowed in GPU programs. So even if there is no such outer loop in the original GPU program, it is often easy to partition data to be processed into chunks, and create such a loop structure. We will see more discussions about this in Section 3.5.1.

Note that we only use Figure 2.4 to illustrate the intuitive idea of CPU-GPU pipelining approach. The actual implementation is more complicated and other factors have been

taken into consideration. For example, in addition to copying the transformation results back to GPU, we may have to copy data from GPU to CPU so that the data arrays to be transformed are up to date if they have been modified in GPU memory. This also makes the overhead control more complicated since we have to monitor the pipelining system status more frequently at the boundaries of data transferring and data transformation. On the other hand, the best number of CPU assistant threads depends on the number CPU cores. If $\Delta = 1$, then the best number of threads to be created should equal to the total number of CPU cores minus 1. If $\Delta = \text{the number of CPU cores} - 1$, then we can create one thread at one time. One CPU core is reserved for the main thread to dispatch GPU tasks and monitor the pipelining system. Since modern CPU processors are equipped with more than one core, this job assigning policy with respect to thread affinity is easy to implement and quite efficient. In both cases, the ultimate purpose is to fully utilize the CPU resources to reduce the total CPU transformation time.

To the best of our knowledge, we are not aware of previous uses of such CPU-GPU pipelining for GPU program optimizations. The pipelining scheme exploits two distinctive features of GPU computing: the presence of massive data-parallelism as we have mentioned earlier, and the independence of the memory systems that CPUs and GPUs work on. The second feature is vital for the data transformations to proceed without interfering the normal execution of the GPU kernel.

The CPU-GPU pipelining scheme trades certain amount of CPU resource for the enhancement of GPU computing efficiency. The usage of the extra CPU resource is not a concern for most GPU applications because during the execution of their GPU kernels, CPUs often remain virtually idle.

2.5.2 Overhead Minimization through LAM

Even though the CPU-GPU pipelining scheme helps overlap the remapping process with computation, it is still important to minimize the overhead of the remapping transformations. Large overhead leads to a deeper pipeline, which in turn causes two consequences.

First, the warm-up stage of the pipeline is long, leaving many initial iterations of a loop unoptimized. Second, many transformation threads must run concurrently, resulting in a large burden to the host system. Further, increasing the pipeline depth cannot always hide the entire transformation overhead. When the number of transformation threads is so large that the capacity of CPU or memory bus is saturated, increasing the pipeline depth can only prolong the transformation time.

As mentioned earlier, between the two mechanisms for realizing thread-data remapping, data layout transformation usually incurs more transformation overhead. This sub-section hence concentrates on overhead reduction for data layout transformation.

High-Level Design We develop an approximation algorithm to save transformation overhead. It aims at striking a good tradeoff between the transformation overhead and the quality of the resulting thread-data mapping. The rationale is that sometimes, sacrificing the optimality of the resulting divergences a little may significantly reduce both the time for layout computation and the number of required data movements.

The developed scheme is named LAM (label-assign-move). Figure 2.5 outlines the algorithm. For easy explanation, the following description assumes that the kernel contains only one condition statement. It is a loop, which has $data[tid]$ iterations in the execution of thread tid . At the end of this section, we discuss the application of the algorithm in a general setting.

The high-level design of LAM is that it partitions the possible values in $data$ into a number of classes, and then constructs a data layout so that most warp segments⁵ are pure—that is, containing elements of the same class. To avoid unnecessary data movements, the scheme first uses a class ID to label each warp segment of the original $data$ array. The elements in a warp segment that belong to its labeled class won't be moved during the construction of the new data layout. By ignoring the differences within a class and avoiding unnecessary movements, LAM may save significant transformation overhead.

⁵A warp segment is a segment of $data$ mapping to a thread warp.

Detailed Algorithm Figure 2.5 outlines the algorithm of LAM. LAM includes three steps as suggested by its name. In the “label” step, it partitions the value range of array *data* to R sub-ranges, represented by r_i ($i = 1, 2, \dots, R$). Each element in *data* must belong to one sub-range (we say that the sub-range covers that element). There are two attributes associated with a sub-range, stored in $r_i.quota$ and $r_i.toFill$. Let n_i represent the number of elements in *data* covered by r_i . The algorithm sets $r_i.quota$ to $\lfloor n_i/warpSize \rfloor$ initially, indicating the largest number of warps that can be entirely covered by r_i . The “label” step examines each warp segment in *data*. Let r_{max} be the sub-range with the largest coverage for a warp segment; the algorithm labels that warp segment with seg_{max} . Suppose an element in that warp segment a is not covered by r_i , the algorithm puts the location of a into the array $r_i.toFill$, indicating that this location should be filled with some other element of *data* that is covered by r_i . An exception occurs when the quota of r_i is used up, when that warp segment is labeled as “mixed”.

The “assign” step creates an array *destLoc* (initial values are all zeros) of the size of *data* to store the desired destination of every element in *data*. It includes two sub-steps. It examines every location in the *toFill* array of every range first, because the data elements in those locations must move. Let $data[i]$ be one of such elements, and l be the value range $data[i]$ falls into. Then, $destLoc[i]$ is set to a location stored in $r[l].toFill$, and the algorithm marks that location as “taken”. An exception happens when all locations in $r[l].toFill$ have been taken. In that case, the algorithm puts i into an array *toMix*, indicating that $data[i]$ needs to be later moved into a to-be-mixed warp segment; the exact destination is yet to be determined. At the end of this sub-step, some locations in some *toFill* arrays may not be taken yet. The second sub-step checks every element in the “mixed” warp segments to see which can be used to fill those remaining openings. As soon as a location in “mixed” warp segments becomes vacant, it is assigned as the destination of an element in the *toMix* array.

When the “label” step finishes, the value of $destLoc[i]$ is the desired destination for $data[i]$ if $destLoc[i]$ is not zero. Otherwise, $data[i]$ needs no movements. The final step,

“move”, simply copies the elements of *data* to *dataCopy* accordingly.

Discussions The description of LAM assumes that the kernel contains only one loop condition statement. If there are more condition statements, the algorithm works in a similar way. The only change is on how the R classes are defined: One option is to consider the path vector of a thread as one point in a K -dimensional space, and define the R classes by clustering the points to R groups.

Divergences on different statements may cause different influence on the overall performance. We may incorporate such differences into LAM by using weighted distances during the clustering process, with weights as the degree of performance influence.

The use of *quota* and the labeling process in LAM ensure that using LAM, every data element in the new layout must have one and only one counterpart in the original layout. This property determines the correctness of the transformation scheme.

A key parameter in LAM is R . The larger it is, the better the resulting mapping is, in terms of eliminated divergences. But meanwhile, more overhead will be incurred by the transformation. In our experiments, we set it to 10. A desirable scheme is to dynamically determine its appropriate value through runtime adaptation, which may happen cooperatively with the adaptation of CPU-GPU pipelin depths (Section 2.5.1). The detail is left for future explorations.

2.6 Evaluation

For the techniques to be useful in practise, they must not only be able to enhance program performance in certain scenarios, but also guarantee no performance degradations in other cases. Our evaluation concentrates on both aspects:

- *Benefits*. How effective are the proposed techniques in removing thread divergences and improving performance?

```

// data[N]: the transformation target
// constants:
// R: # of val partitions;
// WZ: the warp size;

“Label” Step:
mn = min (data); mx = max (data);
createValRngs (R, mn, mx, r);
calQuota (data, r);

for (warp=0; warp < N/WZ; warp++){
  i = warp*WZ;
  l = argmax (coverage(r[k], data[(i+WZ-1)]));
  k
  if (r[l].quota >0) {
    r[l].quota--;
    label[warp] = l; // label this warp
    for (j=i; j < i+WZ; j++) {
      if (data[j] ≠ r[l]){
        // a to-be-filled location
        r[l].toFill [r[l].ttl++] = j;
      }
    }
  }
  else
    label[warp] = MIXED; // -1
}

“Assign” Step:
// for elements in toFill locations
toMixN = 0; k = 0;
for (i=0; i < R; i++){
  for (j=0; j < r[i].ttl; j++){
    orgLoc = r[i].toFill[j];
    l = getRng (data[orgLoc]);
    if (r[l].cur < r[l].ttl) { // not full yet
      // put to a to-be-pure warp
      destLoc[orgLoc] = r[l].toFill[ r[l].cur ];
      r[l].cur++;
    }
    else // go to a mixed warp
      toMix [ toMixN++ ] = orgLoc;
  }
}
// for other elements
for each “MIXED” warp w {
  for (j = w*WZ; j < w*WZ+WZ; j++){
    l = getRng (data[j]);
    if (r[l].cur < r[l].ttl) {
      // fill an opening in a to-be-pure warp
      destLoc [j] = r[l].toFill[cur];
      // use a to-be-mixed element to fill the left opening
      destLoc [toMix[ k++]] = j;}}

“Move” Step:
// dataCopy is a copy of data created already
for (i=0; i < N; i++)
  if (destLoc[i]) dataCopy [destLoc[i]] = data[i];

```

Figure 2.5: The approximation scheme, LAM (label-assign-move), for reducing the overhead in data layout transformation.

- *Overhead.* How effective are the techniques in reducing and hiding transformation overhead? Can they prevent the transformations from degrading the performance of the application, even in extreme scenarios where no potential benefits exist for the transformation?

2.6.1 Methodology

Table 5.1 shows the five benchmarks selected for evaluating the techniques in both aspects. The first two come from real applications, and the other three come from the NVIDIA CUDA SDK [1]. These benchmarks contain different amount of thread divergence and hence different potential gain for the transformation to produce. The first four benchmarks have a number of thread divergences, suitable for the assessment of the effectiveness in thread-divergence removal. The last benchmark, even though containing condition control flows, has no thread divergences. We include it to test whether the proposed techniques can ensure the basic efficiency of the program in this extreme case.

As an echo to previous observations [11,24], one of the difficulties we come across in the

Table 2.1: Benchmarks for divergence removal experiments

Program	Description	Lines of code	Cause of divergence	Diverg. ratio*
3D-lbm	lattice Boltzmann model for partial differential equation	3380	condition	50-100%
gafort	Fitness calculation in a genetic algorithm	3723	condition & loop	75%
marching-Cubes	Geometric isosurface extraction	2178	condition	99%
reduction	Parallel reduction	1264	condition	100%
black-scholes	Option pricing	501	none	0

* Divergence ratio: the number of diverging thread warps over the total number of warps.

evaluation process is the lack of standard GPU benchmarks. Many previous studies have used only some kernels in industry released SDKs (e.g. NVIDIA CUDA SDK). However, because those kernels are created partially to show off the appealing power of GPU, most of them have virtually no GPU-unfriendly features—such as thread divergence⁶. Even for programmers of real GPU applications, as they are told that thread divergence is a GPU performance killer, they typically avoid using GPU if the program includes complex control flows. The consequence is the sparsity of interesting applications for the evaluation of thread divergence removal, even in real application suites. The implication of such a phenomenon is not that thread divergence elimination is unimportant, but the opposite: The current exploitation of GPU is evidentially limited by the weakness in handling thread divergences; resolving such a weakness may well extend the use of GPU for high performance computing.

Our experimental platform is an NVIDIA Tesla 1060 hosted in a dual-socket quad-core

⁶The program *reduction* used in the experiment is a version NVIDIA uses to explain GPU performance hazards in their tutorials.

Intel Xeon E5540 machine as the platform. The Tesla 1060 includes a single chip with 240 cores, organized in 30 Streaming multiprocessors. We use CUDA as the programming model.

In our experiments, the transformations are conducted in a semi-automatic manner. We implement a prototype library that includes a set of functions for facilitating both the profiling and the transformations. These functions include the LAM scheme, the functions for profiling divergences and condition variable values, and for finding optimal thread-data mappings. For each program, we manually insert the invocation of those functions on appropriate variables in the CPU-GPU pipelining fashion. As the focus of this work is on the examination of the effectiveness and feasibility of the transformation techniques, we leave their integration in compilers to the future.

In the rest of this section, we report the performance gain and the transformation overhead of every benchmark. We summarize the results at the end of this section.

2.6.2 3D-LBM

The program, 3D-LBM, is a PDE (partial differential equation) solver based on the LBM (lattice Boltzmann model), implemented by Zhao [78] for GPU. The LBM is a model initially designed to solve fluid dynamics through the construction of simplified microscopic kinetic models. Its extended version may help solve the parabolic diffusion equation, a critical component in the elliptic Laplace and Poisson equations, widely used in the manipulation of images, surfaces and volumetric data sets. The developed LBM program is for 3-D fluid simulation. The code has been highly optimized, appearing to be an efficient alternative to traditional implicit iterative solvers on CPU [78].

Our experiment concentrates on a kernel named “streamAndCollision”, one of the most time-consuming kernels. In that kernel, each thread works on a fluid node. It checks 19 directions in a 3D space, and performs density calculation and node update if it finds neighboring nodes in the a direction. As shown in Table 2.2, this condition statement causes 50–100% thread warps to diverge, depending on the thread block size.

Table 2.2: Experimental results of 3D-LBM

Block size		8	16	32
Diverg. ratio	org	50%	100%	100%
	opt	0%	0%	0%
Exe. time (μ s)	org	4627	4951	5601
	opt	3961	3360	3901
Speedup		1.17	1.47	1.44

org: original program. opt: the optimized program with thread-data remapping and efficiency control applied.

As the neighboring nodes are input-specific and compile-time unknown, the divergences can only be handled through runtime transformations. In our experiment, our tool chooses data layout transformation to apply because of the use of global memory in the GPU kernel and the satisfaction of the conditions listed in Section 2.4.3.

Table 2.2 reports the experimental results. Because the program performance is sensitive to the size of a thread block, we experiment with three different block sizes, ranging from 8 to 32 (the largest size is 32 because of the limits on the 3D space). Half to all of the 1024 warps diverge on the condition statement. The transformation removes all divergences, making the kernel run 1.17 to 1.44 times faster than the original version does. The transformation overhead is 1900μ s, smaller than the length of an invocation of the kernel, hence completely hidden by the CPU-GPU pipelining (with pipeline depth as 1).

2.6.3 GAFORT

GAFORT is a CUDA program derived from an OpenMP version in SPEC OMP3.2. It computes the global maximum fitness in a genetic algorithm. The program starts with an initial population and then generates children who go through crossover, jump mutation, and creep mutation with certain probabilities. All of these major computations are implemented as GPU kernels. In each iteration, some runtime generated random numbers determine which operation(s) among crossover, jump mutation and mutation needs to be done by a child. The random numbers hence introduce misalignments of control flow into the GPU kernels. The random number generator runs on CPU, interleaved with the GPU kernels.

The optimized kernel is the children generation kernel. Each thread works on one child gene. It first determines whether a crossover operation needs to be applied to the child gene. The criterion is based on both the fitness of every candidate genes and a crossover probability between every randomly picked pairs. If a crossover is necessary, the thread conducts such operation on the child gene.

The kernel contains two types of divergences. The first is due to the different decisions on whether a crossover is to be performed on the children genes. The second happens in the crossover computation. That computation contains a loop, whose trip-count equals the length of the segment of parent genes the child needs to use, which differs across children genes. We handle the two types of divergences by combining them into one: The case of no crossover operations is equivalent to that the trip-count of the crossover loop is zero.

Similar to *3D-LBM*, the GPU kernel of this program uses global memory for most arrays and meet the conditions listed in Section 2.4.3. The tool hence selects data layout transformation to apply. A direct application of the data layout transformation would add $8000\mu\text{s}$ overhead, largely offsetting the benefits it can bring. With the LAM algorithm and CPU-GPU pipelining, the overhead can be completely hid.

Table 2.3 reports the results. Unlike *3D-LBM*, this program appears insensitive to the thread block size. In all the cases, the transformation reduces the divergence ratio from 75% to the minimum, 67%. The minimum divergence ratio is still significant because of the large variations in the trip-counts of the crossover loop. Besides the reduction of thread divergences, the transformation on this kernel produces a side effect that it reduces the distance among array elements accessed by adjacent threads, which increases the coalesced global memory accesses. Overall, the transformation yields 1.24 speedup to the kernel.

2.6.4 MarchingCubes

The Marching Cube Isosurface application is from the CUDA SDK, which extracts a geometric isosurface from a volume dataset using a marching cubes algorithm. In this program, a mathematical function is used to create a 3D grid, which is then loaded into the GPU

Table 2.3: Experimental results of GAFORT

Block size		32	64	128	256
Diverg. ratio	org	75%	75%	75%	75%
	opt	67%	67%	67%	67%
Exe. time (μ s)	org	67751	67902	67751	67786
	opt	54772	54778	54756	54724
Speedup		1.24	1.24	1.24	1.24

Table 2.4: Experiment results of MarchingCubes

Block size		32	64	256	512
Diverg. ratio	org	99%	99%	99%	99%
	opt	98%	83%	67%	34%
Exe. time (<i>mus</i>)	org	9264	9147	9132	9579
	opt	6155	6926	4672	3718
Speedup		1.51	1.32	1.95	2.58

device memory. After the isosurface triangles have been calculated, the graph is rendered immediately. This program provides an interactive GUI interface, by which users can perform a variety of actions, such as rotation, zoom in, and recomputation of the isosurface. The program invokes some kernels corresponding to the user’s input. One of the frequently used kernels is the triangle generation kernel. It is a kernel for graphics rendering. The kernel runs only on the occupied voxels by looking up in a table stored in 1D texture memories. Thread divergences occur because of the non-uniform distribution of the occupied voxels.

Different from the previous two benchmarks, the kernel in this program uses texture memory. As mentioned in Section 2.4.3, reference redirection is an appealing option in such a scenario and is selected to apply.

Table 2.4 presents the experimental results. The benefits from the transformation depends on the size of a thread block. The divergence ratio of the optimized version ranges from 34% to 98%, and the speedup ranges from a factor of 1.51 to 2.58. The larger the block size is, the more significant the benefits are. This is because the thread-data remapping is among the threads within a thread block. As the block size grows, the effects of the remapping become more clear.

Table 2.5: Experiment results of Reduction

Input size		2^{21}	2^{22}	2^{23}	2^{24}	2^{25}
Diverg. ratio*	org	100%	100%	100%	100%	100%
	opt	50%	50%	50%	50%	50%
Exe. time (μ s)	org	1010	1626	2007	2851	3986
	opt	927	1474	1788	2565	3549
Speedup		1.09	1.10	1.12	1.11	1.12

2.6.5 Reduction

This benchmark is an implementation of the parallel reduction included in the NVIDIA SDK. It computes the sum of an array through a tree-based approach. The SDK contains multiple versions of the implementation. For our evaluation purpose, the version we use is the one containing interesting thread divergences. At each level of the tree, only part of the threads get involved in the computation. For instance, on the first level of the reduction tree, a condition check on whether the thread ID “tid” is an even number. If so, the thread conducts summation of two elements in the array; otherwise, it does nothing. This condition check causes divergences to every thread warp.

According the principles in Section 2.4.3, for this program, the reference redirection is selected as the transformation technique because its influence is on the computation but not on the global memory references. The transformation follows the description in Section 2.4.1. It includes the creation of a new array, “newIDArray”, the first half of whose elements are assigned with even values ranging from 0 to the thread block size, and odd values for the second half. In the kernel, the references to “tid” are replaced with “newIDArray[tid]” (except in the memory loading statement). After such a transformation, on the first level of the reduction tree, all of the first half of the threads in a block get involved in the calculation, hence all of the thread divergences are removed. There are still some divergences on the other levels of the reduction tree. However, as the first level covers a large portion of the kernel running time, the performance improvement is evidential as shown in Table 2.5 on input arrays of different sizes.

We note that as shown in the NVIDIA tutorial, the thread divergences in this pro-

gram can be completely removed through algorithm changes, leading to further speedup. However, unlike the transformations conducted in our experiment, such changes require programmer's domain knowledge and complete manual efforts. And that approach is specific to the reduction problem, rather than a generally applicable, potentially automatic solution to thread divergence elimination.

2.6.6 BlackScholes

The Black-Scholes model is widely used in the pricing of options in financial engineering. This program from NVIDIA SDK shows an implementation of the model in CUDA for European options.

The kernel contains a condition statement in the polynomial approximation of cumulative normal distribution function, "cndGPU". But profiling results show that no warps diverge on that condition statement. Applying the thread-data remapping transformation to such a program can yield no benefits but possible slowdown for the extra overhead. We use this benchmark as an extreme case to examine whether the transformations can guarantee the basic efficiency of GPU programs. This examination has its practical values. Even though it may be possible to automatically rule out the use of the transformations for some GPU applications containing no divergences, it is not always easy to do: For some applications, the number of divergences may depend on the input data sets. The guarantee of basic efficiency is important for the application of the transformations in such scenarios.

The thread-data remapping step on the CPU computes the values of the condition variable but does not relocate any data because the condition variable values suggest no need for that. The remapping step takes over 10 times of the kernel execution time to run. However, with the efficiency control, the remapping is shut down automatically as soon as the GPU kernel finishes. So, the execution time of the kernel shows no noticeable changes compared to those of the original kernel and program (as shown in Figure 2.6), indicating the effectiveness of the scheme in hiding the overhead and protecting the basic efficiency of the program.

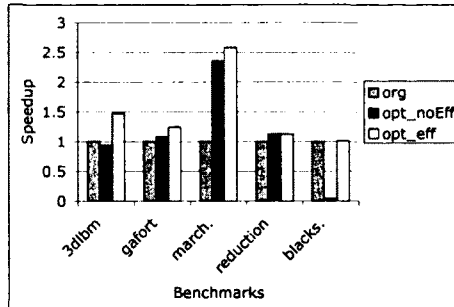


Figure 2.6: Speedup with and without efficiency control.

2.6.7 Summary of Experimental Results

To provide a holistic view of the experimental results, we put the speedup results of all benchmarks into Figure 2.6. For each program, we select the result of a most representative case.

The three bars of a benchmark in Figure 2.6 correspond to the normalized execution times of three versions of the kernel: the original version, the version after the thread-data remapping transformation but without efficiency control, and the version after the transformation with efficiency control. (The baseline of the normalization is the execution time of the original version.)

The large speedups of the third version demonstrate that the transformation generates significant performance benefits by reducing thread divergences in the kernels. The comparison with the performance of the second version reveals that the efficiency control techniques are crucial for the benefits to show up by effectively hiding the transformation overhead. The *blackscholes* results demonstrate the importance of the efficiency control in protecting the basic efficiency of the program in extreme cases. Overall, the results demonstrate the effectiveness and feasibility of the proposed techniques in handling thread divergences, one of the critical performance hazards in GPU computing.

2.7 Related Work

There is a body of work focused on the optimization of some specific GPU applications (e.g., [19, 45, 68]), building general-purpose GPU performance tuning tools [39, 51], or optimizing memory references [6, 36, 67]. Our discussion concentrates on the studies closely related to thread-divergence elimination and data transformations.

Fung and others [24] have tried to reduce thread divergences through special hardware extensions. Our techniques do not need special hardware support. Carrillo and others [11] have proposed loop splitting and branch splitting for optimizing GPU applications that contain loops and branches. Their techniques aim at the alleviation of register pressure, rather than the reduction of thread divergences. The goal of our work is complementary to theirs.

Data layout transformation has been used to reduce cache and TLB misses in CPU [14, 16, 17, 25]. We are not aware of previous uses of the transformation for thread divergence elimination in GPU, which features some special challenges, especially those caused by the distinctive properties of GPU thread divergences and their implications, and the conflict among the large overhead of the transformation, the need for the transformation to occur on the fly, and the little tolerance of overhead in the massively parallel computing on GPU.

2.8 Summary

This chapter describes a systematic exploration in using runtime thread-data remapping to eliminate thread divergences in GPU computing. It proposes the use of reference redirection and data layout transformation for the realization of thread-data mappings. It characterizes the constraints, weaknesses and strengths of the two mechanisms by analyzing the novel implications that GPU computing imposes on the uses of both mechanisms. It presents a CPU-GPU pipelining scheme and a LAM algorithm to effectively hide and reduce thread-data remapping overhead. Together, these techniques remove significant numbers of thread divergences for a set of GPU applications, creating up to 1.47 times of speedup, demon-

strating the feasibility and large potential of runtime thread divergence elimination, and opening up opportunities for streamlining GPU applications on the fly.

Chapter 3

A Unified Framework for Both Control and Memory Irregularities

3.1 Motivation

The massive parallelism of GPU is embodied by the equipment of a number of streaming multiprocessors (SM), with each containing dozens of cores. Correspondingly, a typical application written in GPU programming models (e.g., CUDA [43] from NVIDIA) creates thousands of parallel threads running on GPU. Each thread has a unique ID, *tid*. These threads are organized into warps¹. Threads in one warp are assigned to a single SM, and proceed in an SIMD (Single Instruction Multiple Data) fashion. As a result, hundreds of threads may be actively running on a GPU at the same time. Parallel execution of such a large number of threads may well exploit the tremendous computing power of GPU, but not for irregular computations.

Dynamic Irregularities in GPU Computing Irregularities in an application may throttle GPU throughput by as much as an order of magnitude. There are two types of irregularities, one on data references, the other on control flows.

¹This work uses NVIDIA CUDA terminology.

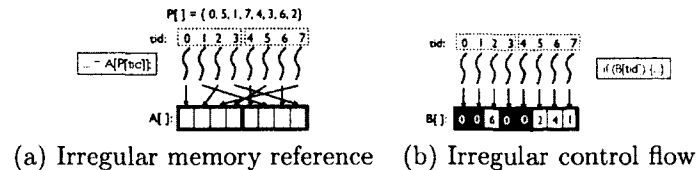


Figure 3.1: Examples of dynamic irregularities (warp size=4; segment size=4). Graph (a) shows that inferior mappings between threads and data locations cause more memory transactions than necessary; graph (b) shows that inferior mappings between threads and data values cause threads in the same warp diverge on the condition.

Before explaining irregular data references, we introduce the properties of GPU memory access. (Without noting, “memory” refers to GPU off-chip global memory.) In a modern GPU device (e.g., NVIDIA Tesla C1060, S1070, C2050, S2070), memory is composed of a large number of continuous segments. The size of each segment is a constant², denoted as Z . One memory transaction can load or store all data in one memory segment. The accesses by a set of threads at one load or store instruction are coalesced into a single memory transaction, if these threads are within a warp and meanwhile the words accessed by them lie in a single memory segment. An *irregular reference* refers to a load or store instruction, at which, the data requested by a warp happens to lie on multiple memory segments, causing more (up to a factor of W ; W for warp size) memory transactions than necessary. Because a memory transaction incurs latency of hundreds of cycles, irregular references often degrade the effective throughput of GPU significantly.

A special class of irregular data references is *dynamic irregular references*, referring to irregular references whose memory access patterns are unknown (or hard to know) until execution time. Figure 3.1 (a) shows an example. The memory access pattern of “A[P[tid]]” is determined by the runtime values of the elements in array P , whose content causes an irregular mapping between threads and the locations of the requested data, resulting in four memory transactions in total, twice of the minimum. Being dynamic, these references are especially hard to tackle, making effective exploitation of GPU difficult for many applications in various domains, including fluid simulation, image reconstruction, dynamic

²In real GPU devices, the value of Z varies across data types. The difference is considered in our implementation but elided in discussions for simplicity.

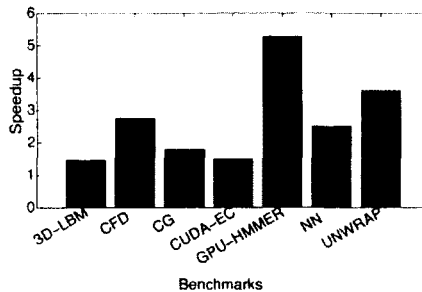


Figure 3.2: Potential performance improvement when dynamic irregularities are eliminated for applications running on a GPU (Tesla 1060).

programming, data mining, and so on [42, 62].

Dynamic irregularities also exist in program control flows, causing *thread divergences* as discussed in Chapter 2. Thread divergences typically happen on a condition statement. When threads in a warp diverge on which branch to take, their parallel execution turns into a serial execution of the threads that take different branches. Figure 3.1 (b) illustrates such an example. Consider the first warp in the graph. Due to the values of the data mapped to the threads, only thread 2 takes the “if” branch. During the execution of that thread, all the other threads in that warp have to stay idle and wait. Note that because the warp is not completely idle, no other warps are allowed to run on that SM during that time, causing waste of computing resource. Consider a typical case where each warp contains 32 threads. The waste of the SM throughput is up to 96% (31/32). The problem is especially serious for loops. Consider a loop “for (i=0; i<A[tid]; i++)” in a kernel and A[0] to A[31] are all zero except that A[13]=100. All threads in the warp have to stay idle until thread 13 finishes the 100th iteration.

Dynamic irregularities severely limit the efficiency of GPU computing for many applications. As shown in Figure 3.2, removing the dynamic irregularities may improve the performance of a set of GPU applications and kernels (detailed in Section 5.5) by a factor of 1.4 to 5.3.

There have been some recent explorations on the irregularity issues. Some propose new hardware features [24, 42, 62], others offer software solutions through compiler techniques [6,

11,36,67,73]. Software solutions, being immediately deployable on real GPUs, are the focus of this work. Previous software solutions mainly concentrate on cases that are amenable to static program analysis. They are not applicable to *dynamic* irregularities, whose patterns remain unknown until execution time. A recent work [74] tackles dynamic divergent control flows, but in a limited setting (as elaborated in Section 5.7).

Overall, a systematic software solution to address *dynamic* irregularities in GPU computing is yet to be developed. In fact, what remains missing are not just solutions, but more fundamentally, a comprehensive understanding to the problem of irregularity removal itself. For instance, as Figure 3.1 shows, the two types of irregularities stem from the relations between GPU threads and runtime data values or layouts, but the relations are preliminarily understood. No answers exist to the questions such as what data layouts and thread-data mappings minimize the irregularities, what the computational complexities are for finding desired layouts or mappings, and how they can be effectively approximated.

Moreover, previous explorations (in software) have treated the two kinds of irregularities separately. But in many real applications, both may exist at the same time and connect with each other—optimizing one may influence the other (e.g., *3dlbm* shown in Section 5.5). It is important to treat them in a holistic fashion to maximize overall performance.

Overview of This Work In this work, we aim to answer these open questions and contribute a comprehensive, practical software solution to both types of dynamic irregularities. First, we unveil some analytical findings on the inherent properties of irregularities in GPU computing. This includes the interactions between irregular control flows and memory references, the NP-completeness of finding the optimal data layouts and thread-job mappings and a set of heuristics-based algorithms, as well as the relations among dynamic irregularities, program data, and GPU threads. These findings substantially enhance the current understanding of the irregularities. Second, we provide a unified framework, named **G-Streamline**, as a comprehensive solution to both types of dynamic irregularities. G-Streamline has several distinctive properties. It is a pure software solution and works on

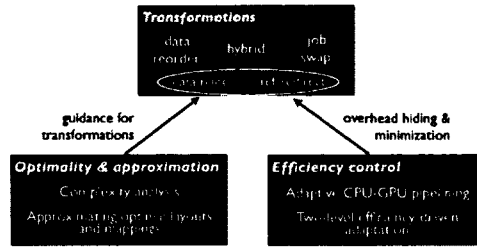


Figure 3.3: Major components of G-Streamline.

the fly, requiring no hardware extensions or offline profiling. It treats both types of irregularities at the same time in a holistic fashion, maximizing the whole-program performance by resolving conflicts among optimizations of multiple irregularities of the same or different types. Its optimization overhead is transparent to GPU executions, jeopardizing no basic efficiency of the GPU application. Finally, it is robust to the presence of various complexities in the GPU application, including the concealing of the data involved in condition statements, the overlapping of the data involved in irregular data references.

We build G-Streamline based on a perspective illustrated in Figure 3.1 (a) and (b): Both irregular memory references and control flows essentially stem from an inferior mapping between threads and data (data locations for the former; data values for the latter). This perspective leads to the basic strategy of G-Streamline for irregularity elimination: enhancing the thread-data mappings on the fly. To make this basic strategy work efficiently, we develop a set of techniques organized in three components as shown in Figure 5.1.

The component, “transformation” (Section 3.3), includes techniques for the realization of new thread-data mappings. Its core consists of two primary mechanisms, data relocation and reference redirection. The former moves data elements on memory to create new data layouts; the latter redirects the references of a thread to new memory locations. Together they lead to three transformation techniques—data reordering, job swapping, hybrid transformation—with respective strengths and weaknesses, suitable for different scenarios. There are two key conditions for the transformations to work effectively: the determination of desirable data layouts or mappings, and the minimization and concealment of transfor-

mation overhead.

The second component, “optimality & approximation” (Section 3.4), helps meet the first condition by answering a series of open questions on the determination of desirable data layouts and mappings for GPU irregularity removal. It proves that finding the optimal data layouts or thread-data mappings in order to minimize the number of memory transactions is NP-complete. For the minimization of thread divergences, it shows that the problem is NP-complete as well but with respect to the number of conditional branches rather than the number of threads. Based on the theoretical insights, this component provides a heuristics-based algorithm for each type of transformations, enabling the computation of near-optimal data layouts or thread-data mappings. Meanwhile, it offers some guidelines for resolving conflicts among the optimizations of different irregularities.

The third component, “efficiency control” (Section 3.5), addresses overhead issues. On one hand, because the irregularities are dynamic, optimizations must happen during run time. On the other hand, transformations for irregularity removal are usually expensive due to the data movements and relevant computations involved. To address that tension, the “efficient control” component employs two techniques. First, based on a previous proposal [74], it adopts an adaptive CPU-GPU pipelining scheme to offload most transformations to CPU so that the transformations can happen asynchronously with the GPU kernel execution. The scheme effectively hides transformation overhead from kernel execution, and meanwhile, protects the basic efficiency of the program by automatically shutting down transformations when necessary. Second, it uses a multilevel adaptation scheme to reduce transformation overhead. The first level is on the tuning of individual transformations; the second level is on the selection of different transformation methods, according to their distinctive properties and the runtime scenarios.

Contributions In summary, this work makes four-fold contributions:

- It provides the first software solution for handling *dynamic* irregularities in both control flows and memory references for GPU computing.

- It proves the computational complexities of irregularity removal, and reveals the essential properties of the irregularities along with their relations with threads and data, advancing current understanding to GPU irregularity removal substantially.
- It develops a set of transformations, analyzes their properties and applicabilities, and proposes several heuristics-based algorithms to circumvent the NP-completeness of irregularity removal.
- It develops a multilevel efficiency-driven adaptation scheme and integrates it into a CPU-GPU pipelining mechanism, demonstrating the feasibility of on-the-fly software irregularity removal solutions.

3.2 Problem Definition

Before describing the three components of G-Streamline, we first present some terms and abstract forms to be used in the following discussions.

Recall that in Section 2.2, a *kernel* is a function executed on GPU. On an invocation of a kernel, thousands of threads are created and execute the same kernel function. They may access different data and behave differently due to the appearances of *tid* in the kernel. Arrays are the major data structure in most GPU kernels, hence the focused data structure in this study. Typically, a GPU kernel takes some arrays as input, conducts certain computations based on their content, and stores results into some other arrays (or scalars) as its final output. We call these arrays *input arrays* and *output arrays* respectively (one array may play both roles).

In the following discussions, we use the abstract form “A[P[tid]]” to represent an irregular reference, and “if (B[tid])” to represent an irregular control flow. The arrays “P” and “B” are both conceptual. In real applications, “P” may appear as an actual input array, or results computed from some input arrays (e.g., “A[X[tid]%2+Y[tid]]”), while, “B” may appear as a logical expression on some input arrays. Using these abstract forms gives conveniences to our discussion, but does not affect the generality of the conclusions (elaborated

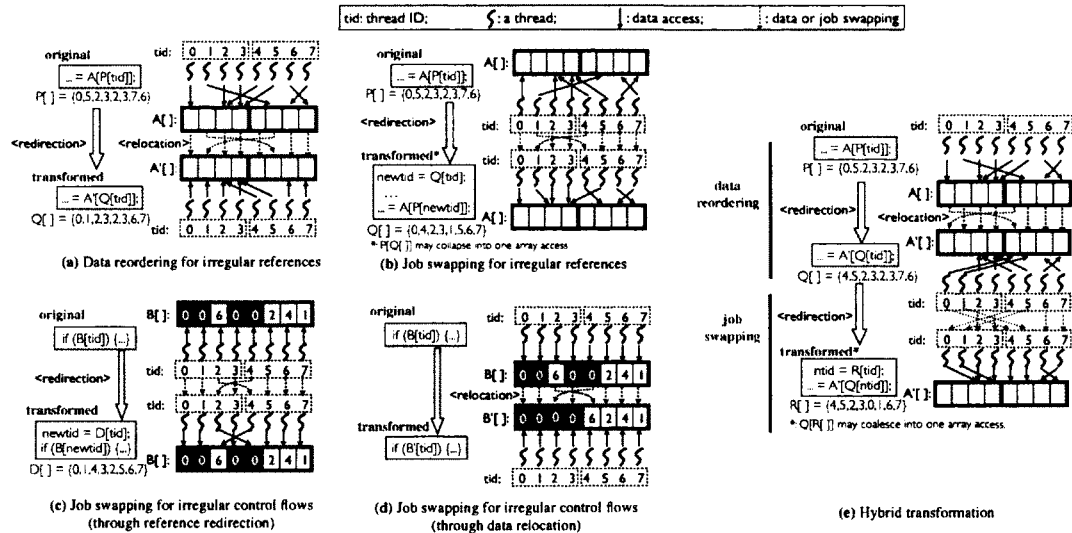


Figure 3.4: Examples for illustrating the uses of data reordering and job swapping for irregularity removal.

in Section 3.6).

3.3 A Framework of Program Level Transformations

G-Streamline contains three main transformation methods for realizing new thread-data mappings. They are all built upon two basic program transformation mechanisms: data relocation and reference redirection. Although the basic mechanisms are classic compilation techniques, it remains preliminarily understood how to use them to remove irregularities in GPU computing—more fundamentally, what are the relations between GPU irregularities and threads and data, how those transformation mechanisms and methods affect the relations, and what the strengths and weaknesses of each transformation method are. This section discusses the mechanisms and transformation methods.

3.3.1 Two Basic Transformation Mechanisms

Data relocation is a transformation that moves data on memory through data copying. It can be either out-of-place (e.g., creating a new array), or in-place (e.g., elements swapping

inside an array).

Reference redirection directs a data reference to certain memory location. In G-Streamline, the redirection is through the use of redirection arrays. For instance, we can replace “ $A[tid]$ ” with “ $A[D[tid]]$ ”; the redirection array “ D ” indicates which element in “ A ” is actually referenced.

3.3.2 Three Transformation Methods

We develop three transformation methods for removing irregular control flows and memory references. Each of them consists of a series of applications of the two basic mechanisms. In the following explanation on how the transformations remove dynamic irregularities, we assume that the desirable mappings between threads and data (locations or values) are known. Section 3.4 discusses how to determine those mappings.

3.3.2.1 Data Reordering

The first strategy is to adjust data locations on memory to create a new order for the elements of an array. Its application involves two steps, as illustrated in Figure 3.4 (a). In the first step, data relocation creates a new array A' that contains the same set of elements as the original array A does but in a different order. The new order is created based on a desirable mapping (obtained from P as shown in Section 3.4) between threads and data locations. In our example, originally, the values of the elements in P cause every warp to reference elements of A on two segments (the top half of the graph). The relocation step switches the locations of $A[5]$ and $A[1]$. The second step of the transformation changes accesses to A in the kernel such that each thread accesses the same data element (likely in a different location) as it does in the original program. The boxes in the left part of Figure 3.4 (a) illustrates the change: $A[P[tid]]$ is replaced with $A[Q[tid]]$, where Q is a newly produced redirection array. After this transformation, all data accessed by the threads in the first warp lie in the first segment; the total needed memory transactions is reduced from four to three. (Section 3.3.2.3 will show how to reduce it further to the minimum.)

Data reordering is applicable to various irregular memory references. But as it maintains the original mapping between threads and data values, it is not applicable to the removal of irregular control flows by itself.

3.3.2.2 Job Swapping

The second method for irregularity removal is exchanging jobs among threads. A *job* in this context refers to the whole set of operations a GPU thread conducts and the entire set of data elements it loads and stores in a kernel execution.

As shown in Figure 3.4 (b), by exchanging the jobs of threads 1 and 4, we make thread 1 access $A[2]$ and thread 4 access $A[5]$. The transformation achieves the same reduction of memory transactions as data reordering does (not reaching the optimal either). When applying job swapping, it is important to keep the integrity of each job—that is, the *entire* jobs of thread 2 and thread 4 in our example must be swapped. To do so, one just need to replace all occurrences of tid in the kernel with a new variable (e.g., $newtid$), and inserting a statement like “ $newtid=Q[tid]$ ” at the beginning of the kernel, where, Q is an array capturing the desired mapping between threads and jobs. The bottom box in Figure 3.4 (b) exemplifies this process. Apparently, the arrays Q and P can collapse into one R such that $R[tid] = P[Q[tid]]$. The collapse may avoid the additional reference “ $newtid=Q[tid]$ ”, introduced by the transformation.

Job swapping is applicable for removals of irregular control flows as well. Figure 3.4 (c) shows an example. In the original program, the values of elements in B cause both warps to diverge on the condition statement. By exchanging the jobs of thread 2 and thread 4, the transformation eliminates divergences of both warps on the condition statement. This example exposes a side effect of job swapping: It may change memory access patterns in the kernel. The swapping in Figure 3.4 (c) impairs the regularity of the accesses to B , causing extra memory transactions. This side effect can be avoided by applying the data reordering transformation described in the prior sub-section as a follow-up transformation to job swapping.

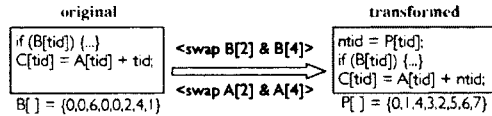


Figure 3.5: Using data relocation for job swapping faces some complexities.

Job swapping can be materialized in two ways. Besides through reference redirection as Figures 3.4 (b) and (c) show, the second way is through data relocation. As shown in Figure 3.4 (d), when the locations of $B[2]$ and $B[4]$ switch while tid remains unchanged in the kernel, threads 2 and 4 automatically swap their jobs. There are some complexities in applying this job swapping method, exemplified by Figure 3.5. First, it requires all input arrays in the kernel (e.g., A and B in Figure 3.5) go through the same data exchanges to maintain the integrity of a job. The incurred data copying may cause large overhead. Second, for this approach to work, it must treat occurrences of tid that are outside array subscripts carefully. For instance, in Figure 3.5, simply switching $A[2]$ and $A[4]$ on memory would cause the expression “ $A[tid]+tid$ ” to produce wrong results. A careful treatment to appearances of “ tid ” that are outside array subscripts can fix the problem, as shown in the transformed code in Figure 3.5 (where, P is an assistant array created to record the mapping between threads and jobs). Finally, at the end of the kernel, the order of the elements in output arrays (e.g., C in Figure 3.5) has to be restored (e.g., switch $C[2]$ and $C[4]$) so that the order of elements match with the output of the original program.

Apparently, relocation-based job swapping applies only to the removal of irregular control flows, but not irregular memory references as the mapping between threads and data locations remains the same as the original.

3.3.2.3 Hybrid Transformations

The third strategy for removing irregularities is to combine data reordering and job swapping. The combination has two benefits. The first has been mentioned in the prior subsection: A follow-up data reordering helps eliminate the side effects that thread divergence elimination imposes on memory references.

The second benefit is that combined transformations often lead to greater reduction of memory transactions than each individual transformation does. As shown in Figure 3.4 (a) and (b), data reordering and job swapping both reduce the needed memory transactions to three for the shown example. Figure 3.4 (e) shows that a combination of the two transformations may reduce the number of memory transactions to two, the minimum. The rationale for the further reduction is that the reordering step creates a data layout that is more amenable for job swapping to function than the original layout is. On the new layout, two threads in warp one reference two data elements in segment two, and meanwhile, two threads in warp two reference two data elements in segment one. Swapping the jobs of the two pairs of threads ensures that the references by each warp fall into one single segment, hence minimizing the number of needed memory transactions.

3.3.2.4 Comparisons

Both types of irregularities may benefit from multiple kinds of transformations. We briefly summarize the properties of the various transformations. Section 3.5 describes the selection scheme adopted in G-Streamline.

Irregular reference removal may benefit from all three strategies (except relocation-based job swapping). Data reordering and job swapping each has some unique applicable scenarios. Suppose the segment size and warp size are both 4. For a reference “A[Q[tid]]” with “Q[] = {0,4,8,12,16,20,24,28}”, data reordering works but job swapping does not; a contrary example is “A[Q[tid]]” with $Q[] = \{0, 1, 2, 5, 2, 5, 6, 7\}$ —no data reordering alone helps as $A[2]$ and $A[5]$ are each accessed by two warps. The hybrid strategy combines the power of the two, having the largest potential. On the aspect of overhead, job swapping incurs the least overhead because unlike the other two strategies, it needs no data movements on memory. In complexity, the hybrid strategy is the most complicated for implementation.

Thread divergence removal relies mainly on job swapping with data reordering as a follow-up remedy for side effects. Between the two ways to realize job swapping, the redirection-based method has lower overhead than the relocation-based method, as by itself,

no data movements are needed. However, that benefit is often offset by its side effect on memory references. On the other hand, the relocation-based method, although having no such side effects, are limited in applicability. Generally, if the data to be moved are accessed by threads in more than one warp, relocation-based job swapping is likely to encounter difficulties. (Consider a modified version of the example in Figure 3.4 (d), where thread 4 originally accesses B[3] rather than B[4].)

Overall, the techniques discussed in this section form a set of options for creating new mappings between threads and data. Next, we discuss what mappings are desirable and how to determine them for the minimization of different types of irregularities.

3.4 Determination of Desirable Data Layouts and Mappings

In this section, we first present some findings and algorithms related to the removal of each individual type of irregularities, and then describe how to treat them when they both exist in a single kernel.

3.4.1 Irregular Memory References

Recall that all three strategies can apply to irregular reference removal. For data reordering, the key is to determine the desirable orders for elements in input arrays; for job swapping, the key is to determine the desirable mappings between threads and jobs; for the hybrid strategy, both data layouts and thread-job mappings are important. We are not aware of any existing solutions to the determination of optimal data layouts or thread-job mappings for irregular reference removal on GPU. In fact, even whether the optimal are feasible to be determined has been an open question.

In this work, by reducing known NP-complete problems, the *3DM* and the *partition problem* [28], we prove that finding optimal data layouts or thread-data mappings is NP-complete for minimizing the number of memory transactions. We will present the proofs and describe the heuristics-based solutions.

3.4.1.1 Data Reordering

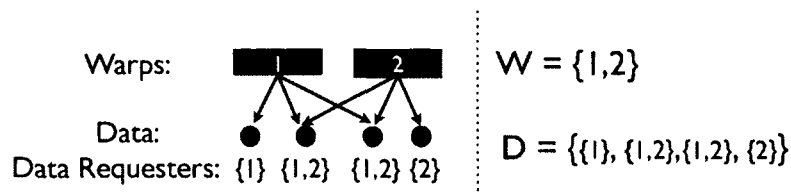
Problem Definition In the Minimal Memory Transaction (MMT) problem, our goal is to group data items into fixed number of memory segments so that the number of memory transactions can be minimized. We define the decision problem version of MMT for NP-completeness proof. Assume there are n threads warps and k data items that need to be grouped into k/m memory segments of size m bytes (k is a factor of m). We define a set W containing n thread warps. There is a set D , a collection of k subsets of the set W , with one subset for each data item. Every subset D_i ($i = 1$ to k) comprises of the integer indices of the warps that request for the i -th data item (as shown in Figure 3.4.1.1). Given a bound B , the question is whether there exists an even partition of the set of data items into $P_1, \dots, P_{k/m}$ such that:

- $P_i \cap P_j = \emptyset$, $|P_i| = m$, and $\bigcup_{i=1}^{k/m} P_i = D$
- Define the weight of P_i as $S(i) = |\bigcup_{D_j \in P_i} D_j|$, and $\sum_{i=1}^{k/m} S_i \leq B$

In the above definition, the k/m disjoint partitions of D represent the partitioning of k data items into k/m memory segments. The weight $S(i)$ for partition P_i , which is the number of unique warps that need to access one or more data items in the partition, stands for the total number of requests for the i -th memory segment from all the warps, which is essentially the number of memory transactions instantiated for the i -th memory segment. Henceforth, the sum of the weights is the total number of memory transactions for all memory segments.

We prove the NP-completeness of MMT decision problem by a polynomial time reduction from 3 Dimensional Matching (3DM). The 3DM problem is proved to be NP-complete and defined as follows.

- Input: Disjoint sets B, G, P , each of size a , and a set of ordered triples $T \subseteq B \times G \times P$
- Question: Does there exist a subset $M \subseteq T$ with $|M| = a$ such that for each pair $(b, g, p), (b', g', p') \in M$ it holds that $b \neq b', g \neq g', p \neq p'$



Proof In the reduction, we generate an special instance of MMT problem from an instance of the 3DM problem. We define the set W of warps in MMT using the set T of triples in 3DM so that $W = T$. Every data item is modeled as a member in B , G or P . Therefore the dataset D corresponds to the set union of B, G, P and $|D| = |B \cup G \cup P|$. The i -th member D_i is a subset of T and includes a triple in T if and only if the i -th element in $B \cup G \cup P$ do not exist in the triple. Then we set $m = 3$ and $B = a \times (|T| - 1)$. Therefore the 3DM problem is reduced to the MMT problem in polynomial time. Next we prove that if there exists such a partition in the MMT problem, there exists a feasible matching M for 3DM problem. The weight of every partition is at least $|T| - 1$, since only when the three elements (b, g, p) associated with three subsets in D form a valid triple in T their corresponding set union miss at most one triple in $|T|$. There are a partitions in total and the sum of the weights is at least $a \times (|T| - 1)$. Hence when the upper bound $B = a \times (|T| - 1)$, there exist a partitioning scheme if and only if every partition contains three data elements (b, g, p) that exist as a valid triple in T and the partitions are disjoint. The MMT decision problem is NP because we can evaluate if a partition satisfies the constrains by set union and other set operations in polynomial time. Therefore we proved the MMT decision problem is NP-complete.

Heuristics Based Algorithm For data reordering, we employ data duplication to circumvent the difficulties in finding optimal data layouts. The idea is simple. At a reference, say $A[P[tid]]$, we create a new copy of A , denoted as A' , such that $A'[i] = A[P[i]]$. Then, we use $A'[tid]$ to replace every appearance of $A[P[tid]]$ in the kernel. With this approach, the number of memory transactions at the reference equals the number of thread warps—

the optimal is achieved. The main drawback of this approach is space overhead: When n threads reference the same item in A , there would be n copies of the item in A' . When there are irregular references to multiple arrays (or multiple references to one array with different reference patterns, e.g., $A[P[tid]]$ versus $A[Q[tid]]$) in the kernel, the approach creates duplications for each of those arrays (or references), hence possibly causing too much space overhead. There are two ways to mitigate the problem. First is to first determine the range of array to be transferred between GPU and CPU which is used as a base reference array for duplication. For example, in $A[P[i]]$, array A is the base array that is used to generate duplicated array. The second way is to dynamically control the amount of duplication so that we will be able to approximate the transformation according to best cost/benefit balance point. Section 3.5 will show how adaptive controls regarding the second approach.

3.4.1.2 Job Swapping

For job swapping, we design a two-step approach. First, consider a case with only one irregular memory reference $A[P[tid]]$. The first step of the approach classifies jobs into M (number of memory segments containing requested items in A) categories; category C_i contains only the jobs that reference the i^{th} requested memory segment of array A . Then for each category (C_i), we put $\lfloor W * \lceil |C_i|/W \rceil \rfloor$ of its members evenly into $\lceil |C_i|/W \rceil$ buckets (W is warp size). This step ensures that each of those job buckets, when assigned to one warp, needs only one memory transaction at $A[P[tid]]$. The remaining jobs of C_i form a residual set, R_i . The second step uses a greedy algorithm to pack the residuals into buckets of size W . Let $\Omega = \{R_i | i = 1, 2, \dots, M\}$. The algorithm works iteratively. In each iteration, it puts the largest residual set in Ω into an empty bucket, and then fills the bucket with some jobs in the smallest residual sets in Ω . It then removes those used jobs from Ω and applies the same algorithm again. This process continues until Ω is empty. This size-based packing helps avoid splitting some residual sets—splits cause jobs accessing the same memory segment to be distributed to different warps, hence incurring extra memory transactions. This job swapping algorithm uses less space than data reordering, but is

mainly applicable for kernels having one or multiple references with a single access pattern (e.g., $A[P[tid]]$ and $B[P[tid]]$). For other cases, G-Streamline favors data reordering.

As the previous section shows, the combined use of data reordering and job swapping may create additional opportunities for optimizations. However, the catch is extra complexities for determining the suitable data layouts and job mappings. A systematic exploration is out of the scope of this work.

3.4.2 Divergent Control Flows

3.4.2.1 Reference Redirection

As Section 3.3 describes, job swapping is applicable for removing irregular control flows. Section 2.4.1 describes reference redirection approach in an elaborate way. The key to its effectiveness is to find a desirable mapping between threads and jobs. Through reducing the *partition problem* [28], we prove that finding optimal thread-job mappings (in terms of the total number of thread divergences) for the removal of irregular control flows is NP-complete with respect to K (K is the number of condition statements in a kernel; assuming each has two branches). The proof is omitted for lack of space.

We extend the algorithms proposed in Section 2.4.1. In the last chapter, we used path-vector-based job regrouping to handle divergences caused by non-loop condition statements. For a kernel with K condition statements, each job has a corresponding K -dimensional vector (called *path vector*), with each member equaling the boolean value on a condition statement. The prior work uses loop trip-count (i.e., number of iterations) based sorting to treat thread divergences caused by a loop termination condition. We have developed a new sorting algorithm that is even quicker than *quick sort* algorithm. The intuition is that since for a lot of threads, the path vectors are the same so during the sorting process, we aggregate the continuously same path vector threads while scanning the array and exchange values. Besides, we put a sentinel (the largest array element in the end) to reduce the total number of instructions and this also helped reduce total sorting time a lot. On the other hand, we handle cases with more than one loop by adding one dimension to the path vector for each

loop. The values in those dimensions are categorized loop trip-counts. The categorization is through distance-based clustering [27]. For instance, for a kernel with two condition statements and one loop whose iterations among all threads fall into L clusters (i.e., 0, 100-200, 1000-1300, >10000), the path vectors of all threads would be in three dimensions; the final dimension is for the loop, and can have only L possible values, corresponding to the L clusters. After integrating loops into path vectors, we can simply assign jobs having the same path vector values to threads in the same warps.

3.4.2.2 Hybrid Approach

The second mechanism is hybrid approach which helps alleviate the side effects on memory reference patterns. It is the approach elaborated in Section 2.4.2, which is referred to as *data layout transformation*. The effect of this approach as if we first apply a thread relocation approach simply by transformation on thread IDs. Then we check the data arrays whose reference pattern has been changed from being coalesced to being random, and transform these data arrays correspondingly to maintain at least the same memory efficiency.

3.4.3 Co-Existence of Irregularities

Irregular control flows and irregular memory references co-exist in some kernels. The co-existence may be inherent in the kernel code, or caused by optimizations as already exemplified in Figure 3.4 (c). As the optimal data layouts or thread-job mappings may differ for the two types of irregularities, the co-existence imposes further challenges to irregularity removal.

G-Streamline circumvents the problem based on the following observation: Even though job swapping affects both control flows and memory references for a thread, data reordering affects only memory references. The corresponding strategy taken by G-Streamline is to first treat irregular control flows using the approach described in the previous sub-section, and then apply data reordering to handle all irregular memory references, including those newly introduced by the treatments to irregular control flows. The handling of irregular

memory references does not jeopardize the optimized control flows.

3.5 Adaptive Efficiency Control

Sophisticated techniques for overhead minimization is important for the optimizations described in this project to work profitably. As dynamic irregularities depend on program inputs and runtime values, transformations for removing them have to happen at run time. These transformations, however, often involve significant overhead. Job swapping, for instance, the most lightweight transformation of the three, requires no data movements, but still involve considerable cost for computing suitable thread-job mappings and the creation of redirection arrays. Without a careful design, the overhead may easily outweigh the optimization benefits.

G-Streamline overcomes the difficulty through a CPU-GPU pipelining scheme, a set of overhead reduction methods, and a suite of runtime adaptive control. These techniques together ensure that the optimizations do not slow down the program in the worst case, and meanwhile, maximize optimization benefits in various scenarios by overlapping transformations with kernel executions, circumventing dependences, and adaptively adjusting transformation parameters. Recall that we have described the basic CPU-GPU pipelining in Section 2.5.1 and we will describe more complicated pipelining schemes in this section.

3.5.1 Dependence and Kernel Splitting

In some programs, the main loop works on a single set of data iteratively; the arrays to be transformed are both read and modified in each iteration of the central loop. These dependences make the CPU-GPU pipelining difficult to apply because the transformation has to happen on the critical path synchronously after each iteration. Transformation overhead becomes part of the execution time, impairing the applicability of the G-Streamline optimizations.

We introduce a technique called *kernel splitting* to solve the problem. The idea is to

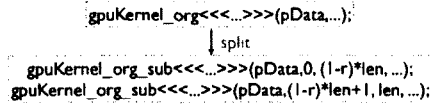


Figure 3.6: Kernel splitting makes CPU-GPU pipelining remain feasible despite loop-carried dependences.

split the execution of a GPU kernel into two by duplicating the kernel call and distributing the tasks. Figure 3.6 shows such an example. In the new program, the invocation of the original kernel “*gpuKernel_org*” is replaced with *gpuKernel_org_sub* and *gpuKernel_opt_sub*. The invocation of the function *gpuKernel_org_sub* behaves the same as the original, but completes only the first $(1 - r)$ portion of the data processed by the original kernel (i.e., the tasks conducted by the first $(1 - r)$ portion of the original GPU threads), while the invocation of function *gpuKernel_opt_sub* completes the remaining tasks. When GPU is executing *gpuKernel_org_sub*, a CPU assistant thread does G-Streamline transformations for the data to be used in *gpuKernel_opt_sub*. Therefore, with the kernel execution split into two, the CPU-GPU pipelining becomes feasible even in the presence of dependences. The rate r is called *optimization ratio*, the determination of which is discussed in Section 3.5.3.

In some of these programs, the suitable data layout and mappings do not vary across iterations. In that case, the analysis for finding the appropriate mappings or data layouts is a one-time operation, and can be put outside of the main loop. But the creation of new arrays have to happen after each iteration of the main loop. For programs having no central loops but multiple phases of computation, the pipelining can still be applied through kernel splitting in the similar way as the previous paragraph describes.

3.5.2 Approximation and Overlapping

In some cases, the overhead of a full transformation is so large that even the pipelining cannot completely hide the overhead. Approximations are necessary to trade optimization quality for efficiency. The partial transformation mentioned in the previous subsection is one example of such approximations. By only transforming part of the data set that is

going to be used in an iteration, the technique reduces transformation time. Even though that technique is described for addressing loop-carried dependences, partial transformation is apparently applicable to all settings regardless of the presence of dependences.

For the elimination of control flow irregularities, we adopt the label-assign-move (LAM) algorithm described in our previous work [74]. The algorithm avoids unnecessary data movements by marking data with a number of class labels and making only necessary switching of data elements such that same classes of data locate adjacently.

An additional technique we use to reduce transformation overhead is to overlap the different parts of a transformation. A transformation usually consists of two steps: producing appropriate data layout or thread-data mappings, copying the produced data to GPU. (For some programs, some data may have to be copied from GPU to host before the transformation.) These steps may all consume considerable time. Our technique treats the to-be-transformed data set as s segments so that the copying of one segment can proceed in parallel with the transformation of another. We call the parameter s the *number of data segments*, determined through the following adaptive control.

3.5.3 Adaptive Control

G-Streamline comes with a multi-level adaptive control that selects the transformation methods and adjusts transformation parameters on the fly.

Coarse-Grained Adaptation The first level of adaptation exists in the CPU-GPU pipelining. As Section 2.5.1 already shows, a transformation shuts down automatically if it runs too slow, and the main CPU thread moves on to the next iteration regardless of whether the transformation finishes. This level of adaptation guarantees the basic efficiency of the program execution.

The second level of adaptation selects appropriate transformation method to use. Recall that irregular reference removal can benefit from different types of transformations. The implementations of these transformations in G-Streamline show the following properties.

Data reordering has the largest space overhead and medium time overhead, but is able to remove all irregular memory references (with data duplication). Job swapping has the smallest overhead in both space and time, but has limited effectiveness and applicability. The strategy in G-Streamline is to use data reordering as the first choice. If its space overhead is intolerable, G-Streamline switches to job swapping. To enable this level of adaptation, multiple copies of the kernel code would need to be created, with each containing the code changes needed for the corresponding transformation. This level of adaptation is optional in the use of G-Streamline.

Fine-Grained Adaptation The third level of adaptation is fine-grained control, which dynamically adjusts the transformation parameters. There are mainly four parameters: the pipeline depth Δ , the optimization ratio r , the number of classes in LAM c , and the number of data segments s .

The pipeline depth Δ (Section 2.5.1) determines the time budget for a transformation to finish. In our implementation, we fix it as 1 but allow multiple threads (depending on the number of available CPU cores) to transform for one chunk of data in parallel. This implementation simplifies thread management.

The number of data segments s (Section 3.5.2) influences the overlapping between transformation and data copying. Its value is 1 by default. In the initial several iterations, if G-Streamline finds that the transformation overhead always exceeds the kernel running time despite what values r takes, it increases this parameter to 5, a value working reasonably well for most benchmarks in our experiments.

The parameters r and c control the amount of work a transformation needs to do. Their determinations are similar. We use r for explanation. We start with the case where no kernel splitting is needed for the target program. A simple way to determine an appropriate value for r is to let its value start with 100%, and decrease by 10% on every failed iteration (i.e., the transformation time exceeds the kernel time). We employ a more sophisticated scheme to accelerate the searching process and meanwhile exert the potential of the transformation

to the largest extent. The scheme consists of three stages as follows:

- *Online Profiling.* This stage happens in the first two iterations of the central loop; r is set to a small initial value (10% in our implementation), represented as r_0 . In the first iteration, the transformation time and the kernel execution time—note, this is the original kernel execution time as no optimizations have been applied yet—are recorded, represented by T_{tr} and T_{org} . If the first iteration fails (i.e., $T_{tr} > T_{org}$), no G-Streamline transformations will be applied to all future iterations. Otherwise, in the second iteration (r_0 of the data to be used have been optimized), the kernel execution time is recorded, represented by T_{opt} . The difference ($T_{org} - T_{opt}$) is the time saved by the optimization, represented by T_{sav} .
- *Estimating Transformation Ratio.* The second stage happens at the beginning of the third iteration. Notice that the desirable value of r , represented as r' , should make the transformation time equal the optimized kernel time—that is, $T'_{tr} = (T_{org} - T'_{sav})$. Assuming that both transformation time and kernel saving time increase proportionally with r , we have $T'_{tr} = T_{tr} * r'/r_0$ and $T'_{sav} = T_{sav} * r'/r_0$. Hence, we get $r' = r_0 * T_{org}/(T_{tr} + T_{sav})$.
- *Dynamic Adjustment.* The third stage adjusts r' through the next few iterations in case that the estimated r' is too large or small. A naive policy for the adjustment is (1) to decrease its value by a step, r_s , on each failed iteration until reaching a success, and (2) to increase its value on each success until a failure, then decrease it by a step size, and stop adjustment. This simple policy is insufficient, illustrated by the following example. Suppose $r_0 = 10\%$, r' equals 30% at the beginning of this stage, and the third iteration is a success. Note that the kernel execution in this iteration actually is on the data optimized in the second iteration, when the optimization ratio is 10% rather than 30%. Therefore, the success of the third iteration does not mean that the transformation of 30% data takes less time than the optimized kernel with 30% as the optimization ratio. In another word, 30% could be too large so that the

fourth iteration (with $r = 30\%$) may fail. The increase of r upon each success in the naive policy is hence inappropriate.

Figure 3.7 shows the adjustment policy in G-Streamline. As the right part of the flow chart indicates, r increases its value on two (rather than one) consecutive successes to avoid the problem mentioned in the previous paragraph. An additional condition for the increase is that the value of r has never been decreased. If r has been decreased, two consecutive successes means that the appropriate value of r has been found (further increase can only cause failures, and further decrease produces less optimization benefits), and the adjustment ends. All future iterations use that r value.

In the case that kernel splitting is needed for dependences carried by the central loop, the dynamic adjustment of r is the same as shown in Figure 3.7 except that the top two boxes on the right are removed. It is due to the fact that the transformed data are used in the current iteration.

In the case that there is no central loop (e.g., *cuda-ec* in Section 5.5), The kernel tasks are split into three parts, executed by three kernel calls. The first part contains 10% of all. During its execution, the CPU transforms 10% of data. After that, the CPU thread uses the measured kernel time T_{org} and the transformation time T_{tr} to estimate what portion (α) of the remaining 90% tasks should run in the second kernel call so that the transformation for the remaining $(1 - \alpha) * 90\%$ tasks can finish before the finish of the second kernel call. The calculation is $\alpha = T_{tr} / (T_{tr} + T_{org})$. The G-Streamline then optimizes for the remaining $(1 - \alpha) * 90\%$ tasks while the second kernel call is working on the $\alpha * 90\%$ tasks. If the second kernel call still finishes early, the transformation is terminated immediately. Otherwise, the third kernel call uses the optimized data to gain speedups.

The size of a data chunk per central-loop iteration may also be adjusted for runtime adaptation. That size influences the length of a GPU kernel invocation, as well as transformation overhead. However, we find it unnecessary to adjust the chunk size given that the transformation parameters in the adaptive control (e.g., the optimization ratio) can already

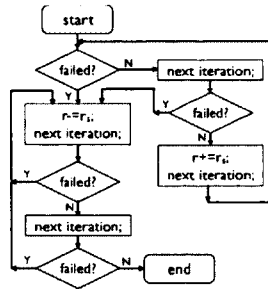


Figure 3.7: Dynamic adjustment for optimization ratio.

alter the rate between transformation overhead and kernel length. In our implementation, the chunk size is the default size in the original program.

3.6 G-Streamline Usage and Other Issues

3.6.1 Overview

G-Streamline is in form of a library, written in C++ and Pthreads, consisting of functions for various transformations (including the heuristics-based algorithms) described in this work, along with the functions for enabling the CPU-GPU pipelining and the adaptation schemes. To activate the pipelined transformations, users need to insert several function calls into the CPU code that encloses GPU kernel invocations. Some minor code changes are necessary to GPU kernels, such as the changes to array reference subscripts as shown in Figure 3.4. Currently, the changes are done manually.

Discussions in this work have been based on the abstract forms of irregular references (“A[P[tid]]”) and condition statements (“if (B[tid])”) defined in Section 3.2. In our experiments, we find that for most applications, “P” and “B” are either some input arrays or results derived by a simple calculation on input arrays. In these cases, their values are easy for G-Streamline to obtain through a simple pre-computation on input arrays before applying the transformations. But in few kernels, the calculations of “P” and “B” are complex. To handle such cases, G-Streamline provides an interface for programmers to provide functions for the attainment of “P” and “B”. For efficiency, the function can produce ap-

proximated values. The calculation of “P” and “B” is part of the transformation process in G-Streamline, and hence can be hidden by the CPU-GPU pipelining scheme and jeopardizes no basic efficiency of the application.

3.6.2 Improved Sorting Efficiency

For divergence path vector based thread relocation algorithm, sorting is an important building block. Either sorting a single array or multiple arrays if the size of divergence path vector is greater than 1 will help generate a near optimal new thread layout. We designed a sorting method that is even faster than *quick sort*, which is the fastest sorting algorithm we have tried among several most commonly used sorting algorithms. The sorting algorithm is called *quicker sort*. The intuitive idea is identify the continuously same data items in the array and treat them as a single data item while scanning and swapping based on the pivot value. The advantage is that we can reduce the total amount of data swapping by grouping the same data items together and later filling them into their corresponding locations. It is typical that the sorting data set has a lot of continuously identical values after several round of recursive quick sort routine. Therefore the quicker sort helps speed up the total thread relocation time. Besides, we set a sentinel element at the end of the array to use the faster quick sort implementation. This quicker sort approach is very important because it helps reduces the CPU transformation time significantly. The CPU transformation intuitively should bear large overhead even compared to the whole GPU execution time. Without the *quicker sort* approach, we found it to be very difficult to gain performance improvement for several benchmarks.

3.6.3 CPU Thread Affinity

Another key factor for making G-Streamline efficient is the CPU helper thread affinity, which means the thread placement on the physical cores. The main thread is responsible for launching GPU kernel calls and spin waiting for monitoring the status of both CPU and GPU threads. The other helper threads are responsible for performing data transformation

or job swapping transformation. The helper threads carry less workload than the main thread. Therefore, the main thread is supposed to get the largest amount of computing resources and the helper threads should only receive computing resource when necessary. We discover that the operating system scheduler does not work well in this scenario. Therefore, we bind every individual CPU thread to a single core and only create the same number of threads as the number of physical cores. This affinity set-up yields the best performance and is crucial to G-streamline overall effectiveness.

The time when the thread affinity should be set also matters. Using *pthread* affinity set functions, we can bind threads to cores in two different ways. The first approach sets thread affinity before the threads are created by specifying thread attribute. The second approach sets thread affinity after the threads are created by inserting a call inside the thread entry function. These two approaches seem to be similar. However, they can make a big difference in performance of G-Streamline. The children threads by default inherit the thread affinity of the parent thread. If inside the thread entry function, the affinity is set as different from its default one, the thread will be migrated to the designated core(s). Although the helper threads need to reside on different cores, at the beginning they reside all on the same core as the parent thread. Since the parent thread starts spin waiting after the helper (also children) threads are created, the parent thread gets more time slices than the children thread with some operating system scheduler. Therefore the parent thread can be a monopoly of the CPU core while the helper transformation threads can't be migrated to other cores and start working in time. We choose the first approach which makes thread immediately run on different cores before the parent thread's spin wait starts. And we observe significant reduction in the transformation overhead.

3.7 Evaluation

We evaluate the effectiveness of G-Streamline on a set of benchmarks shown in Table 4.1. We select them because they contain some non-trivial dynamic irregularities. The bench-

Table 3.1: Benchmarks for G-Streamline and dynamically determined optimization parameters

sloc: source lines of code; r: optimization ratio; s: num of data segments in one transformation

Benchmark	Source	Description	Irreg.	Input	sloc	added sloc		r	s
						all	new		
3dlbm	real app. [78]	partial diff. equation solver	div & mem	32x32x32 lattice	1.7k	200	50	1	1
cfid	Rodinia [13]	grid finite volume solver	mem	800k mesh	0.6k	550	200	0.37	5
cg	NAS (rewritten [36])	conjugate gradient method	mem	75k array	1.2k	250	200	0.3	5
cuda-ec	Tesla Bio [63]	sequence error correction	div	1.1M DNA seq.	2.5k	900	150	0.65	1
gpu-hmmer	Tesla Bio [63]	protein sequence alignment	div	0.2M protein seq.	28k	350	100	1	1
nn	Rodinia [13]	nearest neighbor cluster	mem	150M data	0.3k	210	150	0.7	1
unwrap	real app. [61]	3-D image reconstruction	mem	512x512 images	1.4k	100	70	1	1

marks come from some real applications [61, 78] and some recently released GPU benchmark collections, including Rodinia [13] and NVIDIA Tesla Bio [63]. One exception is *cg*, a kernel derived from an OpenMP program in the NAS suite [5]. Including it is for a direct comparison with a prior study [36] that has optimized the program intensively.

The seven benchmarks cover a variety of domains, and have different numbers and types of irregularities. The program *3dlbm* contain both diverging branches and irregular memory references. Four of the others have irregular memory references, and the other two contains only thread divergences. Together they make a mixed set for the evaluation of not only the various transformations in G-Streamline but also its adaptation schemes. The original implementation of these programs are in CUDA. Previous documents have shown that they have gone through carefully tuning and outperformed their CPU counterparts substantially (e.g., 10–467x for *3dlbm* [78], 20x for *cuda-ec* [63], 9–30x for *cfid* [13]). The inputs to these programs are shown in Table 4.1, some of which (e.g., the input to *cfid*) are directly obtained

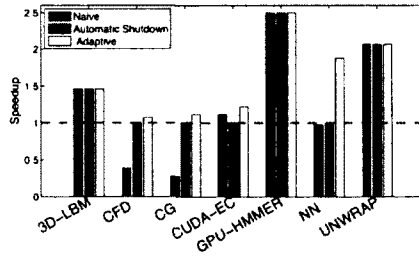


Figure 3.8: Speedup from thread-data remapping.

Table 3.2: Numbers of thread divergences and memory transactions on one GPU SM reported by hardware performance counters

org: original; opt (div): divergence eliminated; opt (both): memory references and divergences optimized.

	3dlbm		cfd		cg		cuda-ec		gpu-hmmmer		nn		unwrap	
	div	mem	div	mem	div	mem	div	mem	div	mem	div	mem	div	mem
org	67k	103M	2.2M	5.2G	0	3.7G	970k	580M	13k	5.6G	0	7.5M	8k	63M
opt (div)	0.5k	90M	-	-	-	-	860k	580M	0.3k	1.8G	-	-	-	-
opt (both)	0.5k	73M	2.2M	4.5G	0	3.0G	-	-	-	-	3	2.5M	8k	13M

from the authors of the benchmarks as the ones coming with the benchmark suite are too small for experiments and typical practical uses.

Our experiments run on an NVIDIA Tesla 1060 hosted in a quad-core Intel Xeon E5540 machine. The Tesla 1060 includes a single chip with 240 cores, organized in 30 streaming multiprocessors (SM). The machine has CUDA 3.0 installed. We analyze performance through CUDA profiler (v3.0.21), a tool from NVIDIA reporting execution information by reading hardware performance counters in one SM of a GPU.

3.7.1 Results Overview

Figure 3.8 reports the speedups of the optimized kernels in the seven benchmarks. The baseline is the execution times of the original kernels; bars higher than 1 means speedup, and slowdown otherwise.

Each program has three bars, respectively corresponding to the performance when the transformations are applied with no adaptation control, with first-level control (i.e., au-

automatic shutdown when transformations last too long), and with all adaptations. The first group of bars indicate that the brute-force application of the transformations, although leading to significant speedup to three programs, cause drastic slowdown to two programs. The first-level adaptive control (automatic shutdown) successfully prevents the slowdown, while the other levels of adaptations yield further substantial performance improvement to most programs. (The reason for the first-level adaptation to throttle speedup of *cuda-ec* is shown in Section 3.7.4.) The benefits of the optimizations are confirmed by the significant reduction of divergences and memory transactions reported by the CUDA profiler, shown in Table 3.2. Overall, the optimizations yield speedups between 1.07 and 2.5, demonstrating the effectiveness of G-Streamline for exerting GPU power for irregular computations. We acknowledge that compared to the data shown in Figure 3.2, some of the results are substantially below the full potential. It is mainly due to the dependences across central loops, transformation overhead, and approximation errors. It indicates possible opportunities for further refinement of G-Streamline.

The software rewriting overhead mainly consists of insertion of G-Streamline library calls for data reordering and threads swapping, customized condition computation functions, and the transformed GPU kernels that optimizes data access patterns and control divergence. Table 4.1 reports the software overhead in terms of the number of lines of inserted code. For most programs, the majority of the inserted code is the duplication of the original code because the new GPU kernels are typically the same as the original except that the thread IDs or array reference IDs are replaced with new IDs. The numbers of lines of newly created code are shown by the “new” column in the table. We acknowledge that the current design of the G-Streamline interface can be further improved to enable more concise expression. Moreover, compiler transformation may further simplify the code changes.

The different degrees of speedups on the seven benchmarks are due to their distinctive features. These programs fall into three categories based on the presence of central loops and dependences. We next discuss each of the benchmarks in further detail.

Table 3.3: Breakdown of time for different memory transaction sizes

	32b-ld	64b-ld	128b-ld	32b-st	64b-st	128b-st	total
org	57M	2M	1M	0	2.5M	0	62.5M
opt	0	10M	0	0	2.5M	0	12.5M

3.7.2 Programs with Independent Loops

Each of the four programs, *unwrap*, *nn*, *3dlbm*, *gpu-hmmmer*, has a central loop with different iterations processing different data sets.

UNWRAP The program, *unwrap*, is for reconstructing 3-D models of biological particles from 2-D microscope photos [61]. Each iteration of the central loop invokes a GPU kernel to transform an image from the Cartesian coordinate system to the Polar coordinate system. In doing so, it accesses the data points in a series of concentric circles, from the image center to the outskirts. The reference patterns lead to inefficient memory accesses.

G-Streamline uses data reordering to optimize the memory accesses. Because the appropriate data layout is determined by the image dimension and typically does not change in the loop, its computation is put outside the central loop. The overhead is completely hidden by the 50 initial iterations of the loop. The creation of new data arrays has to happen in every iteration. The corresponding G-Streamline function call is put inside the loop, working in the CPU-GPU pipelining fashion. The array creation overhead is completed hidden by the execution of the GPU kernels.

As Table 3.2 shows, the transformation reduces the numbers of memory transactions by over 77%. Table 3.3 explains the reduction by showing the breakdown of different sizes of memory transactions. (In the GPU, data can be accessed in 32B-, 64B-, or 128B- segments with the same time overhead.) After putting data accessed by the same warp close on memory, the optimization aggregates many small transactions into some large ones, hence reducing the total number of transactions significantly, cutting execution time by half.

NN The nearest neighbor application, *nn*, finds the k-nearest neighbors from an unstructured data set. Each iteration of the central loop reads in a set of records, computes the

Euclidean distances from the target latitude and longitude. The master thread evaluates and updates the k nearest neighbors. We optimized the read accesses to the unstructured data set through data reordering. We used both the distance computation kernel and the data transfer from host to device to hide the transformation overhead. As Figure 3.8 shows, the overhead for optimizing the whole kernel run can't be completely hidden. Using the adaptive scheme, we were able to achieve a speedup of about 1.8 with the automatically selected optimization ratio equaling 0.7.

3DLBM The program, *3dlbm*, is a partial differential equation solver based on the lattice Boltzmann model (LBM) [78]. It contains both divergences and irregular memory references. Thread divergences mainly come from conditional node updates. The memory reference patterns in the kernel depend on the dimensions of the GPU thread blocks. A previous study [74] has showed up to 47% speedup. But it concentrates on the removal of thread divergences and uses ad-hoc transformations to resolve memory issues. In this work, we apply G-Streamline to the program and achieves a similar degree of speedup. The follow-up data reordering transformation successfully cuts both the newly introduced irregular references and the originally existing ones. The number of memory transactions reduces by over 74%. Both analysis and transformations happen asynchronously outside the main loop because the order does not need to change across iterations.

GPU-HMMER The application *gpu-hmmemr* is a GPU-based implementation of the HMMER protein sequence analysis suite, which is a suite of programs that uses Hidden Markov Models (HMMs) to describe the profile of a multiple sequence alignment. Thread divergences due to the different lengths of protein sequences impairs the program performance. We remove the divergence by job swapping. We replace the original thread-id with re-ordered thread-id except that the thread-ids used in read/write accesses of intermediate result arrays remain unchanged because it hurts no correctness of the program and keeps memory accesses regular. As Table 3.2 shows, the elimination of thread divergences happen to reduce the number of memory transactions as well, indicating that as threads work in a

more coordinate way, they fetch data more efficiently than before. We obtain a speedup of 2.5. The thread-data remapping overhead is completely hidden by the kernel executions in the central loop.

3.7.3 Programs with Loop-Carried Dependences

Two programs, *cf**d*, and *cg*, belong to this category. The iterations of their central loops work on the same set of data iteratively; the computing results of an earlier iteration influence the data to be read by the later iterations. Kernel splitting and multi-segment data transformation ($s = 5$) are applied to both of them.

CFD The program, *cf**d* is an unstructured grid finite volume solver for three-dimensional Euler equations for compressible flow [13]. The inefficient memory references come from the reading of the features of neighboring elements of a node in the unstructured grid of the solver.

The appropriate data layout is loop-invariant and is computed outside the central loop by G-Streamline, while the new array creation has to happen in each iteration. With kernel split, the runtime adaptation of G-Streamline finds that optimization of 37% array elements is appropriate. The optimization yields 7% performance improvement.

CG The program, *cg*, is a Conjugate Gradient benchmark [5]. Lee and others [36] have shown that careful optimizations are necessary when translating *cg* from an OpenMP version to GPU code because of its irregular memory references. They demonstrate that static compiler-based techniques may coalesce some static irregular references in its kernel and achieve substantial performance improvement. But they give no solution to the dynamic irregular references to a vector in its sparse matrix vector multiplication kernel. The vector is read and modified in each iteration, causing loop-carried dependence.

G-Streamline tackles those remaining irregular references by applying data reordering transformation to the vector. The analysis step resides outside of the main loop as the suitable data order does not vary. But the transformation step is in each iteration. G-

Streamline decides on 30% data transformation, and produces 12% further performance improvement over the version optimized through the previous technique [36].

3.7.4 Program with No Central Loop

The program, *cuda-ec*, is a parallel error correction tool for short DNA sequence reads. It contains no central loop, but several kernel function calls. We optimize the main kernel *fix_errors1* by removing divergence through job swapping. As Figure 3.8 shows, the simple application of optimizations without adaptations yields speedup of 1.12. The simple adaptive scheme with automatic shutdown turns off optimizations by default because it cannot tell whether the transformation is beneficial for lack of central loops. G-Streamline, equipped with the complete adaptive control, is able to use the split kernels to estimate optimization ratio (following the scheme described at the end of Section 3.5.3) such that the transformations can overlap with partial kernel executions. The estimated optimization ratio is 0.65, yielding a speedup of 1.22.

3.8 Related Work

Several previous studies have proposed hardware extensions for reducing the influence of irregular memory references or control flows on GPU program performance. Meng and others [42] introduce dynamic warp subdivision to divide a warp so that diverging threads can execute in an interleaving manner. Tarjan and others [62] propose adaptive slip to allow a subset of threads to continue while other threads in the same warp are waiting for memory. Fung and others [24] try to reduce thread divergences through dynamic warp formation. These hardware approaches have shown promising simulation results. As a pure software solution, our approaches are immediately deployable on current real GPU systems.

In software optimizations, the work closest to this study is our previous study on thread divergence removal [74]. We show that some thread divergences can be removed through runtime optimizations with the support of a CPU-GPU pipeline scheme. This work is en-

lightened by that study, but differs from it in several major aspects. First, the previous study tackles only thread divergences, while this study shows that it is important to treat thread divergences with irregular memory references at the same time because of their strong connections. We provide a systematic way to tackle both types of irregularities in a holistic manner, including novel techniques stimulated by the distinctive properties of dynamic irregular memory references on GPU. Second, we contribute some in-depth understanding of the inherent properties of irregularity removal, including the NP-completeness of the problems and the approximation algorithms. They substantially enhance current understanding of GPU irregularity removal. Third, even though the previous study has used reference redirection and data relocation for removing thread divergences, our work reveals the full spectrum of transformations that can be constructed from the two basic mechanisms, and uncovers the properties of each type of transformations. Finally, our work develops some novel efficiency-driven adaptations. Together, these innovations advance state of the art of GPU irregularity removal in both theoretical and empirical aspects.

Another work on thread divergences is from Carrillo and others [11]. They use loop splitting and branch splitting in order to alleviate register pressure caused by diverging branches, rather than to reduce thread divergences.

There have been a number of studies on optimizing GPU memory references. The compiler by Yang and others [73] optimizes memory references that are amenable for static transformations. Lee and others [36] show the capability of an openMP-to-CUDA compiler for optimizing memory references during the translation process. Baskaran and others [6] use a polyhedral compiler model to optimize affine memory references in regular loops. Ueng and others [67] show the use of annotations for optimize memory references through shared memory. Ryoo and others [50] demonstrate the potential of certain manual transformations.

All those studies have shown effectiveness, but mostly for references whose access patterns are known at compile time. To the best of our knowledge, this current study is the first that tackles *dynamic* irregular memory references. Its distinctive on-the-fly transformations are complementary to prior static code optimizations.

An orthogonal direction for enhancing GPU program performance is through auto-tuning tools [39, 49]. The combination of dynamic irregularity removal and auto-tuning may offer some special optimization opportunities.

In CPU program optimizations, data relocation and reference redirection have been exploited for improving data locality and hence cache and TLB usage (e.g., [2, 14, 17]). As a massively parallel architecture, GPU display different memory access properties from CPU, triggering the new set of innovations in this work on both complexity analysis and transformation techniques.

3.9 Summary

In this work, we have described a set of new findings and techniques for the removal of dynamic irregularities in GPU computing. The findings include the interactions between irregular control flows and memory references, the complexity in determining optimal thread-data mappings, a set of approximation algorithms, and the relations among dynamic irregularities, program data, and GPU threads. These findings substantially enhance the current understanding to GPU dynamic irregularities. Meanwhile, we develop a practical framework, G-Streamline. It consists of a set of transformations and adaptive controls for effectively removing dynamic irregularities from GPU applications. G-Streamline works on the fly, requiring no hardware extensions or offline profiling. It treats both irregular memory references and control flows at the same time in a holistic fashion, maximizing the whole-program performance by resolving conflicts among optimizations. Together, the findings and techniques open up many new opportunities for scientific applications involving complex data references or control flows to effectively benefit from massively parallel architectures.

Non-Uniform Cache Sharing Hierarchies

4.1 Overview

One of the features that distinguish modern Chip Multiprocessors (CMP) from traditional processors is the presence of cache sharing among multiple computing units on a chip. The sharing reduces communication latency among co-running threads, but also results in cache conflicts and contention among threads. On a system with multiple chips, the sharing further shows non-uniformity: Cores across chips typically do not share cache as the cores in a chip do.

Researchers have recognized the importance of an effective use of shared cache and developed a number of techniques to exploit it. For example, cache-sharing-aware scheduling in operating systems (OS) research has shown that by assigning suitable programs or threads onto the same chip, one can alleviate the cache contention among co-runners (processes or threads running on sibling cores) and reduce inter-thread communication latency, improving program performance considerably. The effectiveness of those techniques has shown on sets of independent jobs [21, 30, 57, 65] as well as parallel threads inside certain classes of single applications [60].

However, in this work, through a systematic measurement, we find that contrary to the commonly perceived significant effects, cache sharing has very limited influence, either

positive or negative, on the performance of the applications in PARSEC—a recently released benchmark suite that “focuses on emerging workloads and was designed to be representative of next-generation shared-memory programs for chip-multiprocessors” [8]. Our experiments show that for those programs, no matter how the threads are placed on cores (they may share the cache in various ways or do not share cache at all), the performance of the programs remains almost the same.

This surprising finding comes from a systematic measurement consisting of thousands of runs, covering various potentially important factors on the levels of programs (number of threads, parallel models, phases, input datasets), OS (thread binding and placement), and architecture (types of CMP and number of cores). It is derived from the measured running times, and confirmed by the low-level performance reported by hardware performance counters.

After conducting a detailed analysis, we find that the fundamental reason for the insignificant influence is that the development and the currently standard compilation of the programs are oblivious to cache sharing, causing a mismatch between the generated programs and the CMP cache architecture. The mismatch shows on three aspects. First, the data sharing among threads in those programs is typically uniform, that is, the amount of data a thread shares with one thread is typically similar to the amount it shares with any other thread. The uniformity mismatches with the non-uniform cache sharing on CMPs, explaining the insensitivity of the program performance on the placement of threads. Second, the accesses to shared cache lines are limited for most of the programs because of the uniform partition of computation and data among threads, explaining the small constructive effects from shared cache. Finally, the working sets of the programs are typically much larger than the shared cache. The difference between the sharing and non-sharing cases in terms of cache size per thread is not enough to make significant changes in cache misses. Cache contention hence show little influence as well.

The second part of this work explores the implications of the observed insignificance. At the first glance, the observation might seem to suggest that exploitation of cache sharing

is unimportant for the executions of the multithreaded applications. But a set of experiments demonstrate the exact opposite conclusion: Exploiting cache sharing has significant potential, but to exert the power, cache-sharing-aware transformations are critical.

In the experiments, we increase the amount of shared data among sibling threads (the threads to run on the same chip) through certain code transformations. The transformations yield non-uniform data sharing among threads, matching with the non-uniform cache sharing on the architecture. The influence of cache sharing becomes much more significant than on the original programs. Appropriate placement of threads on cores cuts over half of cache misses and improves performance by up to 36%, compared to other placements and the original programs.

To the best of our knowledge, this work is the first that *systematically* examines the influence of cache sharing in modern CMP on the performance of *contemporary multithreaded* applications. Many previous explorations [21, 30, 31, 57, 65] are concentrated on co-runs of independent programs, on which, cache contention is the single main influence by shared cache. The studies on multithreaded programs have been focused on certain aspects of CMP, rather than a systematic measurement of the influence from cache sharing. For instance, many of them have used simulators rather than real machines; some [66] have used old benchmark suites (e.g., SPLASH-2 [71]), or have concentrated on a specific class of applications, such as server programs [60]; some [38] have used old CMP machines with no shared cache equipped. These limitations may not be critical for the particular focus of the previous research—in fact, sometimes they are unavoidable (e.g., using simulators for cache design). However, they may cause biases to a comprehensive understanding of the influence of cache sharing on program performance. As far as we know, none of the previous explorations has included the many factors as covered in this work. These differences explain the departure between the observations made in this work and the previous.

Similar to the observation made by Sarkar and Tullsen [52], we have found only a small number of studies [34, 44, 52] on exploiting *program transformations* for the improvement of shared cache usage (a clear contrast to the large body of work in OS and architecture areas.)

With the importance of program transformations demonstrated in this work, hopefully more research efforts will be triggered in this direction.

In summary, this work consists of three-fold contributions.

- We conduct a systematic measurement on the influence of cache sharing in modern CMP on the performance of contemporary multithreaded applications with seven factors on three levels (program, OS, architecture) considered. The measurement reveals novel observations on the influence of cache sharing.
- We uncover the reasons for the insignificant influence of cache sharing on the multithreaded applications, pointing out that their mismatch with the underlying CMP cache architecture is the main obstacle for exerting the potential of shared cache.
- Through a set of experiments, we demonstrate the potential of cache-sharing-aware program transformations, and conclude that program transformations are the key for exerting the power of shared-cache management (e.g., shared-cache-aware scheduling).

In the rest of this chapter, we describe the design of the measurement in Section 4.2, report the measurement results and findings in Section 4.3, present the exploration on cache-sharing-aware transformation in Section 4.4, discuss related work in Section 4.5, and conclude the paper in Section 4.6.

4.2 Experiment Design

In this section, we first introduce the benchmark suite we use, then present the factors that we vary in the measurement and the corresponding rationales, and finally describe the schemes used for the measurement of times and hardware performance.

4.2.1 Benchmarks

We use PARSEC [8] as the benchmark suite. It is a recently released suite designed for CMP research. The suite includes emerging applications in recognition, mining and synthesis, as

Table 4.1: Benchmarks for non-uniform shared cache experiments

Program	Description	Parallelism	Working Set
Blackscholes	Black-Scholes data		2MB
	diff-eqtn		
Bodytrack	body tracking data		8MB
Canneal	sim. annealing unstruct.		256MB
Dedup	stream com- pipeline		256MB
	pression		
Facesim	face simulation data		256MB
Ferret	image search pipeline		64MB
Fluidanimate	fluid dynamics data		64MB
Streamcluster	online cluster- data		16MB
	ing		
Swaptions	portfolio pric- data		512KB
	ing		
X264	video encoding pipeline		16MB

*: see [8] for detail.

well as systems applications that mimic large-scale multithreaded commercial programs. Studies [7, 8] have shown that the suite covers a wide range of working set sizes, and a variety of locality patterns, data sharing, synchronization, and off-chip traffic, making it an attractive choice over some old parallel benchmark suites such as SPLASH-2 [71]. Table 4.1 lists the 10 programs we use with the working set sizes (on *simlarge* inputs). Programs *dedup* and *ferret* both use the pipeline parallelization model with a dedicated pool of threads for each pipeline stage. Programs *facesim*, *fluidanimate*, and *streamcluster* have streaming behavior. Other programs are data-level parallel programs with different amount and patterns of synchronizations and inter-thread communications. We are unable to use two other programs, *vips* and *freqmine*, because we had difficulty in binding the threads in those programs with processors. All the programs we use are written in Pthreads API. All employ standard Pthreads schemes (locks and barriers) for synchronizations, except *canneal*, which uses an aggressive synchronization strategy based on data race recovery.

Table 4.2: Dimensions covered in the measurement for shared cache experiments

Dimension	Variations	Description
benchmarks	10	from PARSEC
inputs	4	<i>simsmall, simmedium, simlarge, native</i>
# of threads*	4	1,2,4,8
parallelism	3	data, pipeline, unstructured
binding	2	yes, no
assignment*	3	thread assignment to cores
platforms	2	Intel Xeon & AMD Opteron
subset of cores	7	the cores a program uses

*: *Dedup* and *Ferret* have more threads and assignments (see Section 4.3.3).

4.2.2 Factors

To achieve a comprehensive understanding on how much cache sharing influence the performance of multithreaded applications, our experiments include a number of factors that are potentially important for the influence. In this section, we briefly describe those factors and the rationale for selecting them. Table 4.2 summarizes the variations of the factors, and Section 4.3 elaborates on the treatment of the factors in the systematic measurement.

The considered factors come from the program, OS, and architecture levels as follows. (Words in bold fonts correspond to the dimensions in Table 4.2.)

- *Program Level* The major factors include the **input** datasets to the program, the **number of threads**, and the **parallel models**. The first two factors determine the working set of a thread and the intensity of cache contention. We use four input datasets included in PARSEC, as listed in Table 4.2 in increasing order of size, and vary the number of threads from one to eight. The third factor, parallel models, determines the patterns of data sharing and computation.
- *OS Level* The major effect from the OS is thread scheduling, which determines the co-runners on a chip. To examine the potential of the scheduling, we avoid using any particular scheduling algorithms. Instead, we experiment with different **thread-**

core assignments to cover different co-running scenarios as detailed in Section 4.3. Because the experiment needs binding threads to cores, we examine the effects of **binding** by comparing to non-binding cases (detailed in Section 4.3.4.)

- *Architecture Level* We use a Dell PowerEdge 2950 server equipped with 2 quad-core Intel Xeon E5310 processors, and a Dell PowerEdge R80 hosting 2 AMD Opteron 2352 processors. The two machines are the representatives of two typical **CMP architectures** on the market. The Intel machine is based on Front-Side-Bus (FSB) with an inclusive cache hierarchy; the AMD machine is a Cache Coherent Non-Uniform Memory Access (ccNUMA) CMP with HyperTransport links and an exclusive cache hierarchy¹. The explorations on both of them may exhibit the impact of architecture features on how cache sharing influences the performance of multithreaded applications. Both machines run Linux 2.6.22 with GCC4.2.1 installed. Table 4.3 reports the detail of the hardware.

When the number of threads is smaller than the total number of cores in a machine (8 in our experiment), the threads may be assigned to different **subsets of cores**. We experiment with up to 7 (depending on the number of threads) different sets to cover most representative sharing scenarios. In the case of 2 threads on the Intel machine, for instance, the sets of cores we use include 2 sibling cores that share cache, 2 non-sibling cores on a single chip which share the same memory-processor bus, and 2 cores residing on different chips. The 4-thread case has 3 corresponding sets. The 8-thread case has only 1 set, the set of all cores.

Program phase changes may affect the measurement results, especially on the measured potential of thread scheduling. We address this factor as described in Section 4.3.2.

¹The new Intel CMP, Nehalem, resembles this AMD architecture but with an inclusive cache hierarchy.

Table 4.3: Configuration of NUCA CMP machines

	CPU	L1	L2	L3	Memory
Intel	Xeon E5310 1.6GHz quad-core	32KB	2x4MB, each shared by 2 cores	None	8GB shared bus
AMD	Opteron 2352 2.1GHz quad-core	64KB-Icache 64KB-Dcache	512KB	2MB shared by 4 cores	8GB ccNUMA

4.2.3 Measurement Schemes

Our measurement concentrates on running times, cache miss rates, and the amount of shared-data accesses. We use the built-in utility HOOKS in the PARSEC suite to measure running times, and employ the Performance Application Programming Interface (PAPI) library [10] to read memory-related hardware performance counters, including cache miss rates, memory bus transactions, and the reads to cache lines in a “shared” state for every thread. (As required by PAPI for thread-level measurement, we set the pthread scheduling scope to “system” in the hardware performance monitoring.)

Each instance of the set of factors listed in Table 4.2 determines a setting of a run. We call such an instance a *configuration*. For each configuration, we conduct 5 to 10 repetitive runs to reduce the interference from random noises. By default, we use the average performance of the repetitive runs; when necessary, we report the variations as well.

4.3 Measurement and Findings

In this section, we report the detail of the experiments, the results, and findings. As the focus of this work is on the performance influence from cache sharing, our experiments center on the comparisons between the sharing and non-sharing cases—that is, when the threads are bound to sibling or non-sibling cores respectively, as shown in Section 4.3.1. To prevent the effects of thread scheduling from blurring the observations, for the sharing case, we also examine the performance difference caused by different assignments of threads on cores, as reported in Section 4.3.2. We describe the results of *dedup* and *ferret* sepa-

rately in Section 4.3.3. They are two typical pipeline programs with task-level parallelism and numerous pipeline stages. Each stage is handled by a pool of threads. Unlike other programs, the interactions among the threads in these two programs exist both within and between stages, requiring a different set of measurements. The other pipeline program, *x264*, behaves like a data-parallel program, with each thread working on an image frame. So we report its results together with the non-pipeline programs.

4.3.1 Sharing Versus Non-Sharing

To study the influence of cache sharing, we compare the sharing case where the threads are bound to sibling cores, and the non-sharing case where the threads run on non-sibling cores. Let a be the number of threads per chip in the sharing case. The average cache size per thread in the sharing case is $1/a$ of the size in the non-sharing case. The reduced size is part of the effects of cache sharing. We will see that the resulting influence on performance is insignificant.

We use two and four threads in the experiments. (We did not use 8 threads as there would be no interesting non-sharing case to compare.) On the AMD machine, because of the quad-core sharing, the two 4-thread cases actually both have some cache sharing: In the 4-thread sharing case, all four threads run on one chip, thus share one cache; In the 4-thread non-sharing case, there are two threads per chip.

When there are more than one way to assign the threads to cores, we pick the most straightforward way. For instance, in the case of 4-thread sharing case on the Intel machine, we assign threads 0 and 1 to two sibling cores and threads 2 and 3 to the other two sibling cores on a chip. Section 4.3.2 will show that other ways of assignments produce similar results.

Figure 4.1 presents the performance comparison. The running time shown by a bar is the running time of the program in the sharing case normalized to the time in the non-sharing case. So, a bar higher than 1 means the contention on the shared cache and memory bus causes slowdown to the program in the sharing case; a bar lower than 1 indicates that

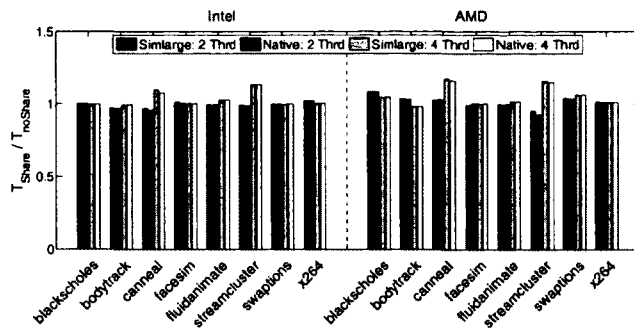


Figure 4.1: The running time of each program in the sharing case normalized to its running time in the non-sharing case. The bars in a group from left to right correspond to the cases of 2 threads on the *simlarge* input, 2 threads on the *native* input, 4 threads on the *simlarge* input, and 4 threads on the *native* input.

the constructive sharing improves the performance of the program. The contention caused by shared cache generates some slowdown to *canneal* and *streamcluster* when the large inputs, *native* inputs, are used, but not much for other programs. On the other hand, the sharing improves the performance of *canneal* and *streamcluster* slightly for *simlarge* inputs, showing that the constructive effects outweigh the cache contention influence when inputs become small. But overall, the sharing shows insignificant influence for most of the programs performance.

The measured cache miss rates further confirm the observed small influence on the performance. Figure 4.2 plots the cache accesses and misses averaged over the threads on the Intel machine for the 2-thread cases on *native* inputs. The cache misses are similar in the sharing and non-sharing scenarios for every program, consistent with the running time results shown in Figure 4.1.

The reasons for the insignificance of the influence come from two aspects. First, the small amount of inter-thread data sharing determines the limited constructive effects from shared cache. Figure 4.3 shows the portion of all the reads on shared cache that happen to access a cache line with a “shared” state (i.e., more than one cores have been reading the data in the cache line.) The larger the portion is, the more data that the co-running threads may prefetch for each other, and hence the more constructive effects cache sharing

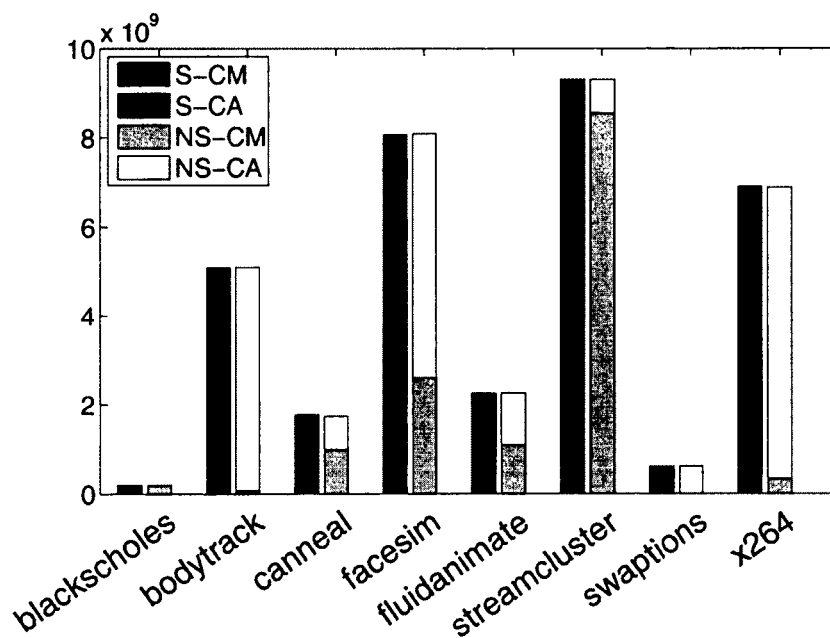


Figure 4.2: Comparisons of L2-cache accesses and misses for 2-thread cases on the Intel machine. (“S” for cache-sharing cases; “NS” for non-cache-sharing cases; “CM” for cache misses; “CA” for cache accesses.)

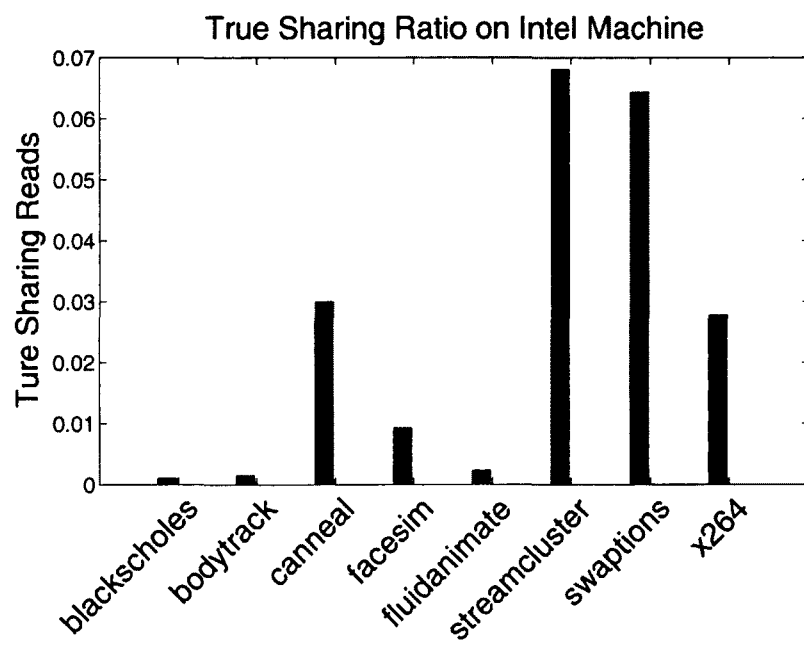


Figure 4.3: Read sharing for 2-thread cases on Intel. It is computed as the number of read accesses to the L2-cache lines with a “shared” state, normalized by the total number of L2-cache accesses.

may impose. The portions are less than 7% for all the programs. Analysis of the source code of the programs confirms the finding. Take the program *canneal* as an example. Each of its threads operates on randomly picked two nodes in a network in every iteration. Because of the large size of the network and the randomness in node selection, it is no surprise to see the small amount of references on shared data blocks.

Second, because the working sets of the programs, as shown in Table 4.1, are typically much larger than the shared cache on a processor, the difference of the cache size per thread between the sharing and non-sharing cases is not enough to make significant changes in cache misses. The cache sharing therefore shows no clear negative effects either. The working set of the program *blackscholes* is smaller than the shared cache on the Intel machine, but it has very few L2 cache line reuses, as shown in Figure 4.2. So the cache sharing has little influence on it either.

4.3.2 Comparisons Among Sharing Cases

The threads in a parallel program usually have certain differences among one another. Threads in a data-level parallel program may compute on different sections of data, resulting in different working sets. Threads in pipeline programs may execute different tasks. In both types of programs, there may be non-uniform communication and data sharing across threads.

In light of the non-uniform cache sharing, the differences among threads may offer opportunities for performance improvement through appropriate placement of threads on cores. The sharing cases considered in the previous subsection contain just one thread-core assignment for each scenario. This section examines the impact that different assignments may have by binding threads to cores in various ways.

For 4-thread cases, we permute the thread-core assignments and exhaust distinctive co-running combinations. For 8-thread cases, we use three representative thread assignments in both the Intel and AMD architectures. We place the threads in such a way that threads whose indices differ by a given distance are assigned to sibling cores. For example on Intel

machine, with the distance set to 1, every two consecutive threads reside on two sibling cores. We vary the distance from 1 to 2 to 4.

Table 4.4 shows the performance difference caused by the different assignments when the *native* inputs are used. Similar results are observed on other input sets. As the table shows, 6 of the 8 benchmarks have less than 5% maximum difference. For the programs that have over 5% differences, *canneal*, *facesim*, we find that the times and cache miss rates for the multiple runs of a fixed *configuration* fluctuate considerably. For instance, five *simlarge* runs of *canneal* on the AMD machine have running times as 0.85, 0.72, 0.83, 0.96, and 0.93. After applying a statistical analysis on the data (Student-distribution with 90% as the confidence value), we observe overlaps of the confidence intervals of different bindings, indicating that the difference in running times is statistically insignificant.

Overall, the different thread-core assignments do not show considerable effects on the program performance. There are two possible reasons. First, the threads in those programs may have similar interactions (communications, synchronizations, etc.) with one another, that is, for each thread, its relations with any other threads may be similar. The second possible reason is program phases. It could be that even though the interactions among threads are not similar among one another, but the interactions show different patterns in different phases of the execution so that no particular assignments work well for all the phases.

We conduct a more detailed experiment to determine the exact reason. We collect the cache miss rates of every 100 million instructions (a typical interval granularity used in phase detection [54, 55]) when the program runs in different thread-core assignments. Figure 4.4 plots the temporal traces of the L2-cache miss rates of *swaptions* and *fluidanimate* when they run on the Intel machine with threads placed on two pairs of sibling cores differently. The three curves in each graph correspond to three sharing cases, in which, the cache on a processor is shared by a different pair of threads. The two programs show different phase change patterns. But on both of them, the three sharing cases show similar L2-cache miss rate curves. Similar phenomena are seen on other programs, indicating that the

Table 4.4: Maximal percentage of the performance differences caused by different bindings of threads to a given set of cores

Benchmarks	AMD		Intel	
	4-t	8-t	4-t	8-t
Blackscholes	0.03	0.01	0.12	0.02
Bodytrack	0.6	0.97	0.64	1.02
Canneal	3.4	7.18	9.34	2.56
Facesim	0.16	11.15	0.43	0.23
Fluidanimate	0.25	0.71	1.23	2.29
Streamcluster	1.88	0.08	0.13	0.05
Swaptions	0.3	1.08	0.1	1.01
X264	0.32	1.12	0.17	0.2

uniform interplay among threads rather than phase changes is the reason for the observed insignificance of the influence of thread-core assignments.

As a side note, the insignificant influence seems to suggest little potential of thread co-scheduling (or thread clustering) for improving the performance of these programs, a contrast to previous results on independent jobs [30,57] and server programs [60]. However, Section 4.4 will show that program transformations would lead to an opposite conclusion.

4.3.3 Pipeline Programs

Unlike data-parallel programs, typical pipeline programs contain numerous concurrent computation stages, and the interactions among the threads exist both within and between stages.

In PARSEC, *ferret* and *dedup* are two such programs. The program *ferret* is a search engine, which finds a set of images that best match a query image by analyzing their contents. The program *dedup* is a program that detects and eliminates redundancy in a data stream. The two programs use a similar producer-consumer model for task parallelism: Every job has to go through several stages before completion; The processing results in one stage is passed to the next stage; Every stage has a dedicated thread pool. The product queue between every two stages is protected by lock-based synchronization schemes. As the programs show similar experimental results, we concentrate on *ferret* for explanation.

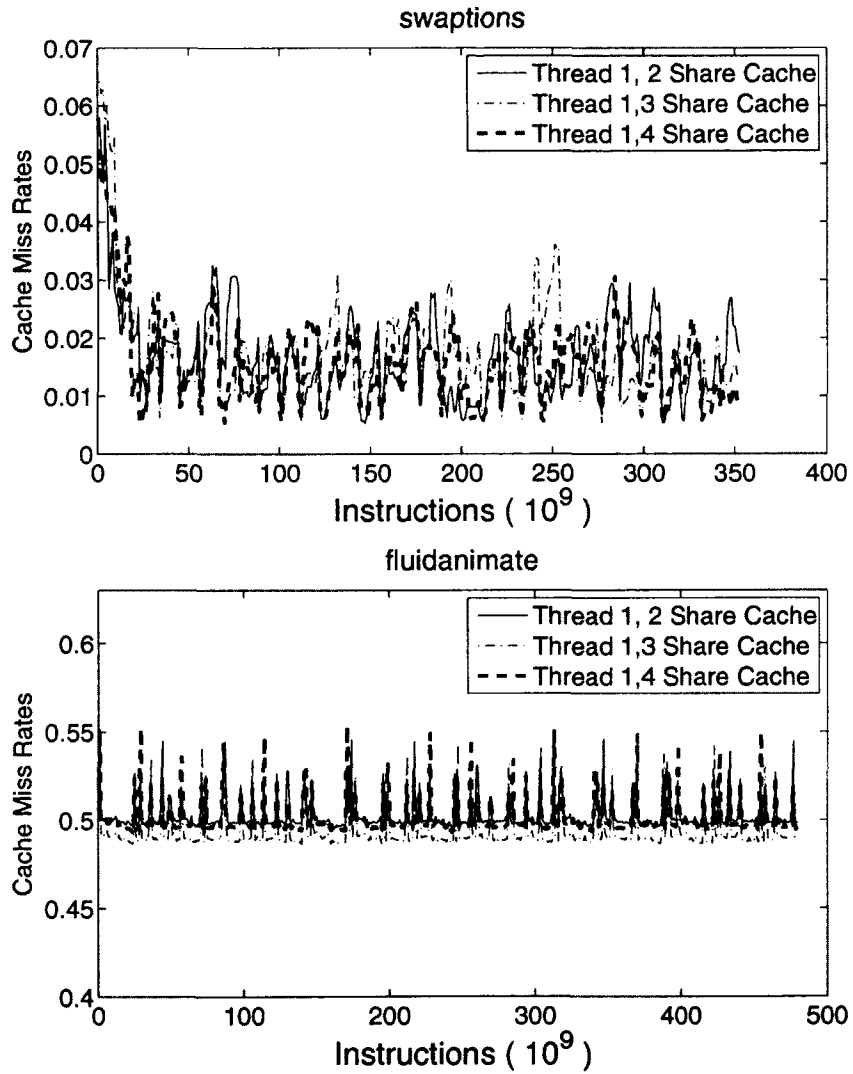


Figure 4.4: Temporal traces of the L2 cache miss rates on the Intel machine when 4 threads are placed on the same set of cores differently.

Table 4.5: Performance of *ferret* on the Intel machine with different thread placement on cores. (S: pipeline stage)

Thread-core Binding of 6 Stages		Time
S1	S2,S3,S4,S5	S6 (s)
0	{0 2 4 6},{0 2 4 6},{1 3 5 7},{1 3 5 7}	1 210.1
0	{0 4 1 5},{2 6 3 7},{0 4 1 5},{2 6 3 7}	2 160.9
0	{1 5 0 4},{3 7 2 6},{1 5 0 4},{3 7 2 6}	3 161.1
0	{0 2 4 6},{1 3 5 7},{0 2 4 6},{1 3 5 7}	1 161.6
0	{0 2 1 3},{4 5 6 7},{0 2 1 3},{4 5 6 7}	4 161.3
No binding		165.7

The program *ferret* has 6 concurrent pipeline stages. The first and final stages are the initialization and completion stages with only one thread in each. The other four stages have the same number of threads. The number is specified in the program input.

Unlike the observations in the previous section, different thread-core assignments for *ferret* sometimes cause significant performance difference. However, the reason for the difference is not the effects from the shared cache, but the load balance across stages. We illustrate the phenomenon by showing the performance of five representative assignments in Table 4.5, each having four threads in every middle stage. Each tuple in the table represents the thread-core assignment in a stage. For example, the first {0,2,4,6} tuple in the top assignment means that the four threads in the second stage are bound to cores 0, 2, 4, 6 in their creation order. (The core layout on the machine is that cores of even index numbers are on one chip and odd numbers on the other.)

Among the five binding cases in Table 4.5, four have about 160s running times, much smaller than the other binding case. What is common about the four good cases is that half of the 8 cores are assigned to stages 1 and 3, and the other cores are assigned to stages 2 and 4. A detailed analysis shows that stages 3 and 4 are the bottleneck, taking more time than other stages. Further experiments confirm that as long as stages 3 and 4 do not share cores, the running times are always about 160s; otherwise, the performance is considerably worse. The non-binding case takes about 160s as well, indicating that the dynamic load balancing in the default Linux scheduler successfully avoids the contention between stages

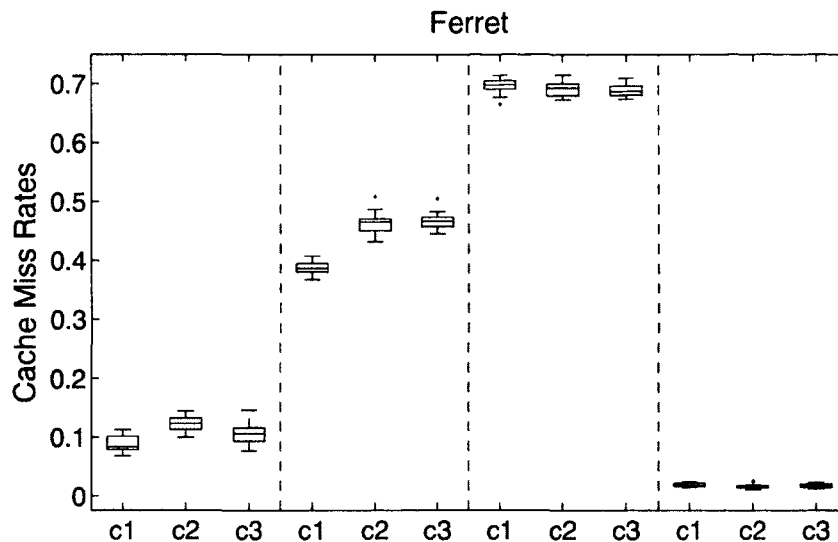


Figure 4.5: Box plot of L2-cache miss rates per thread on the Intel machine when different thread-core assignments are used. (c1,c2,c3 refer to the top 3 configurations in Table 4.5.)

3 and 4.

The cache miss rate results further confirm that the shared cache is not the main reason for the performance difference. Figure 4.5 shows the L2-cache miss rates on the Intel machine when we run *ferret* using the top 3 assignments listed in Table 4.5. Every box in the plot presents the distribution of the cache miss rates of the threads in a stage in 5 runs. The results show that even though the cache miss rates in the different stages differ significantly, the different thread-core assignments impose minor influence on the cache miss rates. In fact, the first configuration (*c1*) shows slightly lower cache miss rates than others, but its performance is the worst, echoing that load balance rather than shared cache is the main factor for such programs.

4.3.4 Effects of Thread Binding

As many of the measurements bind threads with cores, in this part, we examine the effect of the binding, showing that binding threads to cores typically does not worsen the program performance on CMP and thus is a valid way for the study of the influence of cache sharing.

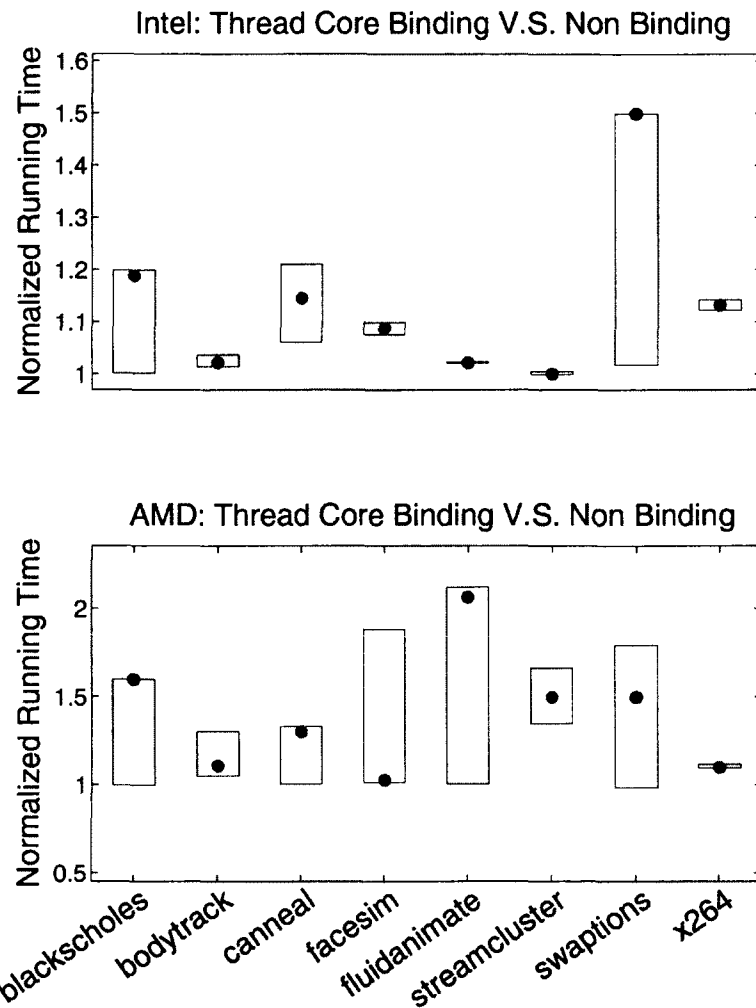


Figure 4.6: Effects of thread-core binding on Intel and AMD machines. Every box shows *min*, *max* and *median* (\cdot) of the five runs for every configuration. The Y-axis is the running time in the non-binding case normalized to the average time in the binding case.

In the binding cases, we bind each thread to a particular core by inserting an invocation of the system function “`pthread_setaffinity_np`” into each benchmark at the point where threads are created. In the non-binding case, we rely on the default Linux scheduler to schedule the threads; the scheduler periodically migrates threads to maintain load balance if necessary. It is important to note that as mentioned in Section 4.2.2, in both binding and non-binding cases, when the number of threads is smaller than the total number of cores in a machine, we use the “`taskset`” command in Linux to specify which set of cores to use.

The results indicate that binding makes the programs perform much more stably than non-binding, reducing performance variations by as much as a factor of 122. We use the average times in the binding case to normalize the running times in the non-binding cases, and plot the 4-thread (2 sibling cores per chip on *native* input) results in Figure 4.6. The heights of the boxes show the large variations of the non-binding running times. Most boxes are above 1, indicating that binding makes the programs run faster than non-binding. We observe the similar phenomena on other configurations, despite the changes in the number of threads, core sets, and inputs.

The observed effects are mainly because the binding reduces cache thrashing and thread migrations. On the other hand, binding may hurt load balance, but that effect is not obvious for those programs due to the uniformity of the threads. This experiment, besides justifying the use of binding in the following explorations, also suggests that, similar to prior observations on traditional SMP (Symmetric multiprocessing) machines, the binding may serve as a strategy for the performance improvement of multithreaded applications running on CMP, despite the presence of shared cache.

Short Summary This section has shown that due to the large working sets and the limited inter-thread data sharing of the multithreaded programs, cache sharing has insignificant (either constructive or destructive) influence on the performance of the programs. Furthermore, we reveal that adjusting the placement of threads on cores has limited potential for performance enhancement of the programs. The main reason is the uniform relations among parallel threads, which mismatches with the non-uniform cache sharing on CMP machines.

These conclusions, drawn from the extensive measurements, appear to hold across inputs, number of threads, sets of cores, and architectures.

4.4 Program-Level Transformation

Although the previous section reports insignificant influence of cache sharing for the performance of PARSEC programs, we maintain that the results do not suggest that cache sharing is a factor ignorable in the optimization of the execution of those programs. The implication is actually the opposite: Cache sharing deserves more attention especially in program transformations.

The conclusion comes from a set of experiments, in which, we transform several programs to make them better match the non-uniform cache sharing on CMPs. Our experiments concentrate on four representative programs. The transformations on them share a single theme, which is to increase the data sharing among sibling threads but not other threads. This section uses *streamcluster* as an example to explain the transformations in detail, and then reports the results on other programs.

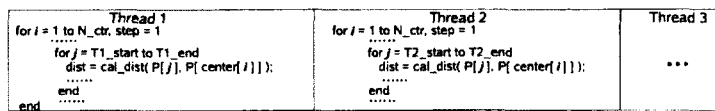
4.4.1 Streamcluster

The program, *streamcluster*, is a data-mining program that clusters a stream of data points. One part of the program takes a chunk of array points and calculates their distances to a center point. This calculation occurs many times and accounts for a major part of the program's running time.

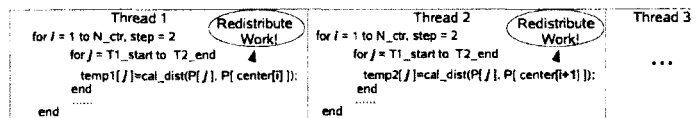
4.4.1.1 Transformation

To highlight the transformation, we use the simplified pseudo-code in Figure 4.7 for the explanation, and assume there are 2 cores per chip.

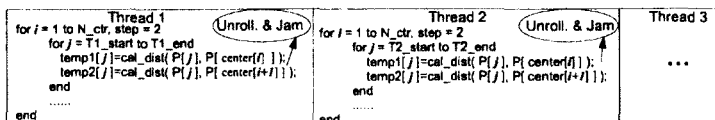
The original version of the program is outlined in Figure 4.7(a). Each of the threads computes the distances of a chunk of data to the center points. The variables *T1.start*,



(a) Original Version (cache-sharing-oblivious)



(b) Cache-sharing-aware transformation. Data sharing increases between sibling threads (e.g. threads 1 & 2), but not across sibling pairs (e.g. threads 2 & 3).



(c) Traditional unroll-and-jam (cache-sharing-oblivious). Intra-thread data locality increases.

Figure 4.7: Simplified pseudo-code illustrating the original and optimized versions of the function *pgain()* in *streamcluster*. It is assumed that two threads constitute a sibling group that share cache.

$T1_end$ represent the start and end of the data chunk assigned for *Thread 1*, $T2_start$, $T2_end$ for *Thread 2*. The outer loop iterates over every candidate cluster center, and the inner loop iterates over every data point in a chunk. The function *calDist* computes the distance between a point and a candidate center.

Figure 4.7(b) illustrates a transformation toward improving the matching between the program and shared cache on CMP. It tries to enhance the data sharing among sibling threads by letting them compute the distances from the same chunk of data points (e.g., thread 1 & 2 on data from $T1_start$ to $T2_end$) to two different center points. The chunk size becomes twice as large as before. The computed distances are stored into two temporary arrays for later uses. (The use of temporary arrays is necessary to circumvent some loop-carried dependencies².) With this transformation, the data sharing among threads becomes non-uniform: For instance, thread 2 shares substantially more data with thread 1 than with thread 3. When sibling threads co-run on a CMP processor, they would form synergistic prefetching with one another. One thread can use the data point brought into the shared cache by the other thread.

We notice that one may improve data locality inside a thread using traditional unroll-and-jam transformation [3]. The transformed code is shown in Figure 4.7(c). (In our implementation, the inner loop is staged to circumvent loop carried dependencies.) In one iteration of the inner loop, each thread computes the distances between a point and two centers, increasing the reuse of the loaded data points. The increase of data reuse is similar to the previous transformation, except that it is inside a thread rather than between threads. The intra-thread and inter-thread transformations are complementary to each other. They can be applied to a program at the same time. In the next section, we report how the inter-thread transformation benefit the program both without and with the intra-thread optimizations.

²Inside the inner loop, after *calDist*, there is an update to a data structure corresponding to the point $P[j]$, which is then used in the computation following the inner loop, causing loop carried dependencies

4.4.1.2 Performance

Figure 4.8 shows the speedup brought by the transformations when four or eight threads run on the Intel machine. In all these runs, we keep sibling threads assigned to adjacent cores. Even though both the inter-thread and intra-thread transformations add extra store operations to the temporary arrays, the results show that their benefits outweigh the overhead substantially. An examination of the source code shows the reason. Each point involved in the distance calculation is of 128 dimensions. As a result, the temporary arrays weight only a small portion of the entire working set.

One may notice that the benefits from the inter-thread transformation is not as significant as those from the intra-thread transformation³. It is because the intra-thread transformation increases the hits in L1 cache, while the inter-thread transformation only benefits L2 usage. However, it is important to note that these two transformations are not competitors. As the "both-share" bars in Figure 4.8 show, based on the code optimized through the intra-thread transformation, the inter-thread transformation further improves the performance by 23%, demonstrating the complementary relations between these two kinds of transformation.

Figure 4.9 reports the normalized L2 cache miss rates and the numbers of memory bus transactions. The performance of the original program is the baseline. In each group of bars, the "inter-share" and "both-share" bars correspond to the cases when the inter-thread transformation is applied without and with the intra-thread transformations respectively. In both cases, the transformation reduces L2 cache miss rates and memory bus transactions substantially, confirming the benefits of the transformation for data locality enhancement despite whether intra-thread optimizations are applied. We stress that the application of the transformations requires the cooperation from thread schedulers. The "inter-noshare" and "both-noshare" bars in Figure 4.9 shows the result when sibling threads are placed on non-sibling cores. The clear contrast with the other bars demonstrates that the shared-

³This result differs from our previous observations [75] because we reimplement the transformation, during which, we manage to remove some inefficiency in the intra-thread transformed code, including the elimination of stores of some intermediate results and some references to assistant data structures.

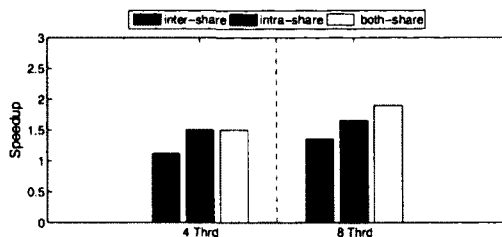


Figure 4.8: Speedup by inter-thread, intra-thread, and combined transformations on the Intel machine. Adjacent two threads share L2 cache.

Table 4.6: Streamcluster with a different input

Data Reuse Level	4	8	16	32	64
Intra (s)	12.2	11.1	10.7	10.9	11.5
Combine (s)	12.0	10.7	10.3	10.3	10.9

cache-aware program transformation creates opportunities to better exert the power of thread co-scheduling or clustering.

We stress that inter-thread sharing is an essential component when using shared cache to improve the program performance by combining the inter-thread sharing and intra-thread sharing transformations together. Simply using intra-thread may not always yield better or equivalent performance compared to using combined transformation. The Table 4.6 shows the streamcluster performance results on a smaller input. We define the data reuse level as the product between the inter-thread sharing level and the intra-thread sharing level, which is the number of times a data point is continuously reused for distance computation in short time interval. As we can see from this table, with the same level of data reuses, the combined transformation case always performs better than the intra-thread sharing case. The reason, as confirmed by L1 cache simulation results, is that increase in intra-thread sharing may increase the chance of cache line conflicts in L1 cache between the data points and center points in the distance calculation. In this case, the center points could have been reused if they had not been evicted out of the cache earlier due to conflicts. The inter-thread sharing case while preserving the same amount of data reuses in last level cache, can help reduce the contention in first level caches. We can control the level of cache conflicts by tuning the

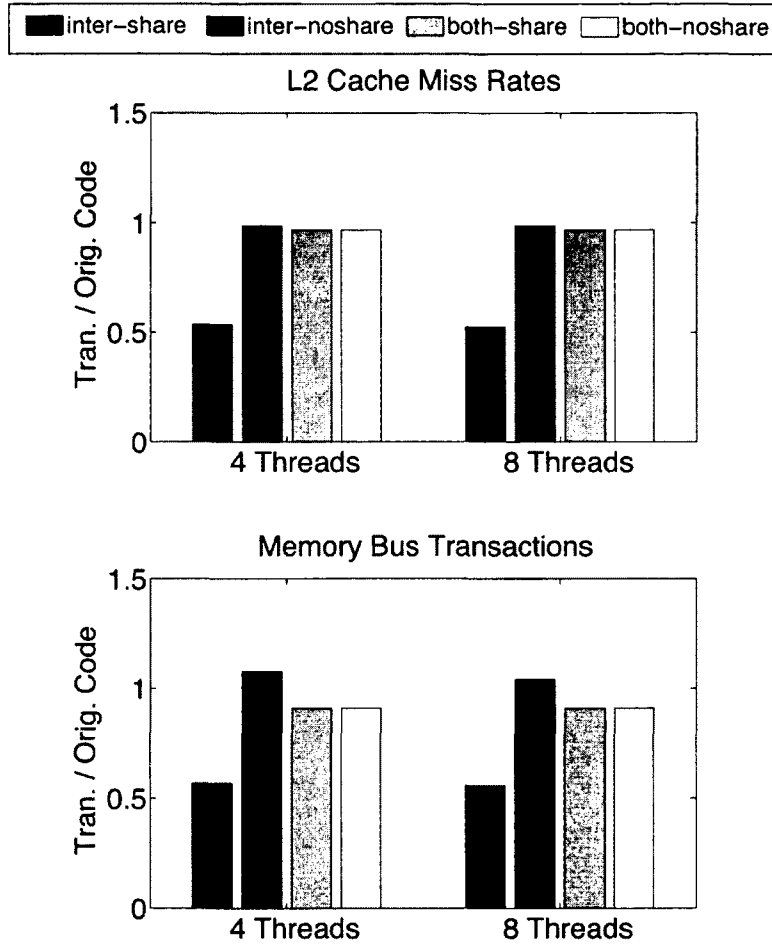


Figure 4.9: The reduction of L2 cache miss rates and memory bus contention on the Intel machine. In each bar group, the first two are when inter-thread transformation is applied to the original program, and the next two are when the combined transformation for both intra-thread and inter-thread sharing is applied. In each pair, the two bars respectively correspond to the cases when the sibling threads of the inter-thread optimized version run on two cores sharing or not sharing cache.

inter-thread sharing level in the combined transformation case, while traditionally exclusive loop unrolling might only deteriorate the performance. Another recently published work [32] corroborates our observation by pointing out the benefits of inter-thread sharing in reducing cache conflicts with a large number of arrays present. In summary, the inter-thread sharing is an essential code transformation technique that should be recognized for performance tuning on the contemporary shared cache architectures.

4.4.2 Blackscholes

The program, *blackscholes*, is a financial application. It calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation. Because there is no close-form expression for the equation, the program uses numerical computation [8].

The input data file of this benchmark includes an array of options. The program computes the price for each of the options based on the five input parameters in the dataset file. The upper bound of the outermost loop in the program controls the number of times the options need to be priced. There are no inherent dependencies between two iterations of the loop. In the original program, the parallelization occurs inside the loop. In each iteration, the options are first evenly partitioned into n (n for the number of threads) chunks. Each chunk is then processed by one thread, which prices the options in the chunk one after one by solving the Black-Scholes equation.

The transformation we apply is similar to the one on *streamcluster*. After the transformation, sibling threads process the same chunk at the same time; their executions correspond to a number of adjacent iterations of the outermost loop.

We observe that the transformation significantly reduces the number of misses on the shared cache on the *native* input, as shown in the left part of Figure 4.10. However, the program running times have no considerable changes. The document of the benchmark (the README file in the package) mentions that “the limiting factor lies with the amount of floating-point calculation a processor can perform.” Through reading the program, we

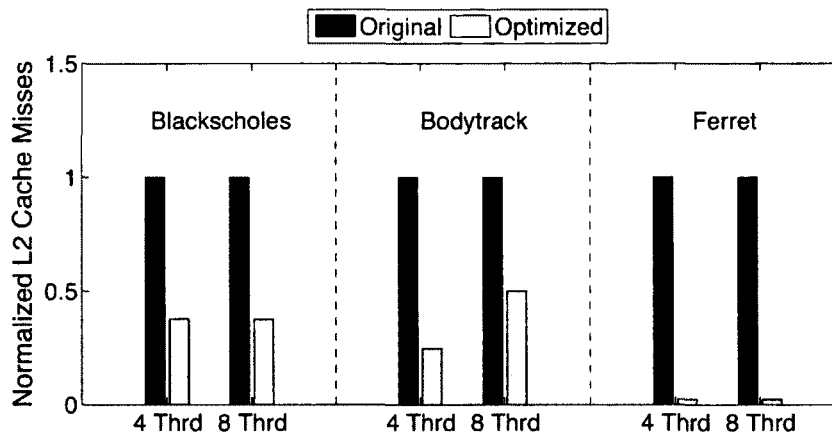


Figure 4.10: The reduction of L2 cache misses of *blackscholes*, *bodytrack* and *ferret* due to cache-sharing-aware transformation. The Intel machine is used.

confirm that the program is a compute-bounded application—after reading an option data, the program conducts a significant amount of computation to solve the Black-Scholes equation with only local variables referenced. For further confirmation, we artificially reduce the amount of computation of the kernel in both the original and optimized programs. The optimized program starts showing clear speedup.

4.4.3 Bodytrack

The program, *bodytrack*, tracks the 3D pose of a human body through an image sequence using multiple cameras. The algorithm uses an annealed particle filter to track the body pose using edges and foreground segmentation as image features, based on a 10 segment 3D kinematic tree body model. The number of particles and annealing layers are supplied as command line arguments.

The program processes frame by frame, and every frame consists of multiple camera images. The program has mainly two parallelized kernels *CreateEdgeMap* and *CalcWeights*. We make sibling cores share workload of the same image and non-sibling cores on different images in the procedure *CreateEdgeMap*, resulting in a 15% speedup with 8 threads pro-

cessing the *native* input on the Intel machine. We also increase the chance of true data sharing for the *CalcWeights* by redistributing the comparison workload for edge maps and foreground segment maps, resulting in a 5% speedup with 8 threads on the Intel machine. The last level cache misses are significantly reduced. We provide the normalized last level cache miss reduction in the middle part of Figure 4.10.

4.4.4 Ferret

The program, *ferret*, is a pipeline program. It implements a search engine for finding a set of images that best match a query image by analyzing their contents. The program contains 6 concurrent pipeline stages. The first and final stages are for initialization and completion with only one thread in each. The other four stages have the same number of threads. The number is specified in the program input.

The fourth stage of the pipeline is the most memory intensive phase of the program. During that stage, the features of a query image extracted by the first three stages are compared against the feature database to identify the top K images closest to the query image. In the original program, each thread processes one query each time. Every query will look up some part of the database according to the locality sensitive hashing (LSH) that maps similar items to the same buckets in the hash table. The part of database that is traversed is much larger than the size of the last-level cache we experiment with, not to mention the size of the whole database.

We apply a two-step transformation to examine the potential of shared-cache-aware program optimizations. First, we split the database into m sections evenly (m equals the number of shared caches in the machine). The threads running on the i th shared cache compare the queries against only the i th database section. As a result, each query is processed by m threads (one on each shared cache). A post step is added to merge the results together. For example, there are eight threads with the first four running on the first chip and the second four on the other chip (assuming one shared cache per chip). After the transformation, the database is split evenly into two sections, owned by each chip. At

most four queries can be processed at one time with threads 0 and 4 processing the first query, threads 1 and 5 processing the second and so on. In the second step, based on their data addresses, we reorder the queries and assign them to the corresponding threads so that nearby queries for the same database section are processed by sibling threads within a short time interval.

This transformation creates a non-uniform relation among threads: Sibling threads share similar data accesses in the same database section, but non-sibling ones do not. Besides the synergistic prefetching among sibling threads, an additional benefit is that shared cache can be used more efficiently: Rather than keeping multiple copies of a database entry in multiple shared caches as the original program entails, the transformed program ensures that different shared caches store different parts of the database. The transformation is insensitive to database size as data reuses are enhanced in an inter-thread level within short time intervals. As shown in the right part of Figure 4.10, the transformation eliminates most shared-cache misses, and yields a speedup of as much as 1.53.

Overall, the experiments demonstrate that after the transformations, cache sharing starts to show its influence, and the placement of threads on cores becomes important for the programs performance. The observations suggest the importance of program-level transformations for improving the usage of shared cache. On the other hand, they further confirm that the uniform relation among threads in the original programs is one of the main causes for the limited influence of cache sharing on performance.

4.5 Related Work

Cache sharing exists in both SMT (Simultaneous Multithreading) and CMP architectures. Its presence has drawn lots of research interest, especially in architecture design and process/thread scheduling in OS.

In architecture research, many studies (e.g., [12, 47, 48, 53, 59]) have proposed different ways to design shared cache to strike a good tradeoff between the destructive and construc-

tive effects of cache sharing. These studies, although containing some examination of the influence of shared cache, mainly focus on the hardware design. Their measurements are on simulators and cover limited factors on the program or OS levels.

In OS research, the main focus on shared cache has been job co-scheduling including thread clustering. Many job co-scheduling studies [20, 21, 30, 31, 57, 58, 65], are on multiprogramming environments, attempting to alleviate shared-cache contention by placing independent jobs appropriately. Some of them include parallel programs in the job set, but the main focus is on inter-program cache contention rather than the influence of shared cache on parallel threads. Tam and others [60] propose thread clustering to group threads that share many data to the same processor through runtime hardware performance monitoring. They concentrate on server programs.

Some studies on workload characterization and performance measurement are relevant to this current work. Bienia and others [7, 8] have shown a detailed exploration of the characterization of the PARSEC benchmark suite on CMP. Because their goal is to expose architecture independent, inherent characteristics of the benchmarks, their measurement runs on *simlarge* input only, and uses a CMP simulator rather than actual machines. Liao and others [38] examine the performance of OpenMP applications on a Sun Fire V490 machine with private cache only. Tuck and Tullsen [66] have measured the performance of SPLASH-2 when 2 threads corun on a SMT processor.

Our work is distinctive in that it examines the influence of cache sharing in CMP on multithreaded programs in a *comprehensive* manner. It explores the numerous factors on program, OS, and architecture levels at the same time, employs modern CMP machines and contemporary multithreaded benchmarks. The systematic examination of the various facets of the problem is vital for avoiding biases that partial explorations may have, and thus improving the understanding of the influence of cache sharing on CMP.

We have found only few studies on exploiting program transformations for the improvement of shared cache usage, which echoes a prior observation [52]. Tullsen and others [34, 52] have proposed compiler techniques to change data and instructions placement

to reduce cache conflicts among independent programs. Nikolopoulos [44] has examined a set of manual code and data transformations for improving shared cache performance on SMT processors. The inter-thread transformation described in Section 4.4 may share some similarity with traditional locality optimization on NUMA architectures for main memory usage [37]. Although both try to redistribute computation and data, our transformation aims to promote synergistic prefetching across threads, rather than increase data accesses to local memory banks.

4.6 Summary

In this work, we conduct a series of experiments on Intel and AMD CMP architectures to systematically examine the influence of cache sharing on the performance of modern multithreaded programs. The experiments cover a spectrum of factors related to shared cache performance on various levels. The multidimensional measurement shows that on both CMP architectures and for all the thread numbers and inputs we use, shared cache on CMP has insignificant influence on the performance of most multithreaded applications in the benchmark suite. The implication, however, is not that cache sharing has no potential to be explored for the execution of such multithreaded programs, but that the current development and compilation of parallel programs must evolve to be cache-sharing-aware. The point is reinforced by three case studies, showing that significant potential exists for program-level transformations to enhance the matching between multithreaded applications and CMP architectures, suggesting the need for further studies on cache-sharing-aware program development and transformations.

Cross-Input Adaptive Performance Tuning for GPUs

5.1 Motivation

Compared to previous approaches for general purpose GPU programming through OpenGL APIs, we have seen significant advance in the accessibility of GPUs for general-purpose computing. One of the most popular programming model is NVIDIA CUDA, as mentioned in previous chapters, abstracts GPU as a general-purpose multithreaded SIMD (single instruction, multiple data) architectural model, and offers a C/C++-like interface—supported by a compiler and a runtime system—for GPU programming. CUDA simplifies the development of GPU programs.

However, developing an *efficient* GPU program remains as challenging as before if not even more. The difficulties come from four aspects. The *first* is the complexity in GPU architecture. On an NVIDIA GeForce 8800 GT, for example, there are over one hundred cores, four types of off-chip memory, hundreds of thousands of registers, and many parameters (e.g., maximum number of threads per block, thread block dimensions) that constrain the programming. The *second* difficulty is that the multi-layered software execution stack makes it difficult to predict the effects of a code optimization. A special difficulty with

CUDA is that currently a GPU program has to be compiled by the NVIDIA CUDA compiler (NVCC) and run on the NVIDIA CUDA runtime system, some details of both of which are not disclosed yet. *Third*, an optimization often has multiple effects, and the optimizations on different parameters often strongly affect each other. *Finally*, some GPU applications are input-sensitive. The best optimizations of an application may be different when different inputs are given to the application. Together, these factors make manual optimizations time consuming and difficult to attain the optimal, and at the same time, form great hurdles to automatic optimizations as well.

On the other hand, optimizations are particularly important for GPU programming. Because of the tremendous computing power of GPU, there can be orders of magnitude performance difference between well optimized and poorly optimized versions of an application [6, 50, 51].

Several recent studies have tried to tackle the problem through empirical search-based approaches. Ryoo and his colleagues [51] have defined *efficiency* and *utilization* models for GPU programs to help prune the optimization space. Baskaran et al. [6] have developed a polyhedral compiler model to optimize global memory accesses in affine loop nests, and used model-driven empirical search to determine the levels of loop unrolling and tiling.

Although both studies have shown promising results, neither of them have explored the influence of program inputs on the optimization. Program inputs refer to both the values and other related properties (e.g., dimensions of an input matrix) of the inputs given to a program. In this work, we initiate an exploration in this new dimension, showing that program inputs may affect the effectiveness of an optimization by up to a factor of 6. Based on the exploration, we develop a tool, G-ADAPT (GPU adaptive optimization framework), to efficiently discover near-optimal decisions for GPU program optimizations, and then, tailor the decisions for each program input.

More specifically, this work makes three major contributions. *First*, we develop a source-to-source compiler-based framework, G-ADAPT, for empirically searching for the best optimizations for GPU applications. The framework is distinctive in that it conducts program

transformations and optimization-space search in a fully automatic fashion, and meanwhile, offers a set of pragmas for programmers to easily incorporate their knowledge into the empirical search process. *Second*, this work examines the influence of program inputs on GPU program optimizations. We are not aware of any previous studies in this direction. The lack of such explorations may be due to a common intuition that as most GPU applications divide a task into small sub-tasks, the changes in their inputs do not matter to the optimizations as long as the sub-tasks remain similar. Our experiments show that, although many GPU kernels conform that intuition, some GPU programs exhibit strong input-sensitivity due to their computation patterns and the interplay with optimization parameters. *Finally*, based on the exposed input sensitivity, we construct a cross-input predictor by employing statistical learning (Regression Trees in particular) to make G-ADAPT automatically tailor optimizations to program inputs. As far as we know, this is the first framework that allows cross-input adaptive optimizations for GPU applications.

Experiments on NVIDIA GeForce 8800 GT GPU show that the adaptive optimization framework can predict the best optimizations for most of the 7 GPU applications with over 93% accuracy. The adaptive optimization improves the program performance by as much as 2.8 times in comparison with manually optimized versions.

We organize the paper as follows. Section 5.2 provides some background on GPU and its programming model. Section 5.3 discusses the challenges in GPU program optimizations. Section 5.4 describes G-ADAPT as our solution to those challenges. Section 5.5 reports evaluation of the framework. Section 5.6 discusses the training overhead and some other complexities of G-ADAPT. After an overview of some related work in Section 5.7, we conclude the paper with a brief summary.

5.2 Experiment Platform

This work uses the NVIDIA GeForce 8800 GT GPU as the architecture. It is a single-chip massively parallel architecture, with 112 cores and 512 MB off-chip memory. The GPU

contains 14 streaming multiprocessors (SMs). Each SM contains 8 streaming processors (SPs) or cores, with the clock rate set at 1.51 GHz. Each SM also includes 2 special function units (SFUs) for the fast execution of complex floating point operations, such as sine, cosine. Besides the computing units, on each SM, there are 8192 32-bit registers and 16 KB shared memory. Unlike cache, the shared memory has to be managed explicitly in each GPU application.

The off-chip memory includes a 512 MB global memory, which is both readable and writable by every SP, and some constant memory and texture memory, which can only be read by the SPs. The constant memory and texture memory are cachable thanks to some on-chip cache, but the global memory is not.

Recall that in Section 2.2, a GPU application written in CUDA is composed of CPU code and GPU kernels. CUDA abstracts the execution of a GPU kernel as multithreaded SIMD computation. The threads are grouped into many warps with 32 threads in each. Those warps are organized into a number of thread blocks. Each time, the runtime system maps one or more thread blocks to an SM. The warps in those blocks are dynamically scheduled to run on the SM. In GeForce 8800 GT, half of a warp is an SIMD execution unit. If one warp is stalled (e.g., due to memory accesses), the other warps can be switched in with nearly zero overhead. Therefore, the number of warps or thread blocks that are mapped to an SM determines the effectiveness of the pipelining execution in hiding latency. As the thread-block size determines the mapping of blocks on SMs, it is an important parameter in GPU program optimizations.

Threads may communicate in the following ways. Threads in a block may communicate through shared memory and be synchronized by a `__syncthreads` primitive. But communications between threads that belong to different thread blocks have to use off-chip global memory; the communications are hence slow and inflexible.

5.3 Challenges for Optimizing GPU Programs

Although CUDA simplifies GPU programming, it reduces little if any difficulty in optimizing GPU applications; to some degree, the added abstractions even complicate the optimization as they make performance prediction still harder.

Optimizations There are mainly two ways to improve the performance of a GPU program: the maximization of the usage of computing units, and the reduction of the number of dynamic instructions. Optimizations to reach the first goal fall into two categories. The first includes those techniques that attempt to increase the occupancy of the computing units. One typical example is to reduce resource consumption of a single thread so that multiple thread blocks can be assigned to an SM at the same time. The multiple blocks may help keep the SM busy when the threads in one block are stalled for synchronization. Example transformations for that purpose include the adjustment of the number of threads per block, and loop tiling. The second category contains the techniques that try to reduce latencies caused by memory references (or branches). Examples include the use of cachable memory (e.g., texture memory), the reduction of bank conflicts in shared memory, and coalesced memory references (i.e., when threads in a warp reference a sequence of contiguous memory addresses at the same time.)

Optimizations to reduce the number of dynamic instructions include many traditional compiler transformations, such as loop unrolling, common subexpression elimination. Although the CUDA compiler, *NVCC*, has implemented many of these techniques, researchers have seen great potential to adjust some of those optimizations, such as the levels of loop unrolling [6, 51].

Challenges It is difficult to analytically determine the best optimizations for a GPU application, for three reasons. *First*, it is often difficult to accurately predict the effects of an optimization on the performance of the GPU application. The effects are often non-linear as what Ryoo et al. have shown [51]. The undisclosed details of the CUDA compiler

and other abstractions add further unpredictability. *Second*, different optimizations often affect each other. Loop unrolling, for example, removes some dynamic instructions and exposes certain opportunities for the instruction scheduler to exploit; but it also increases register pressure for each thread. Given that the number of registers in an SM is limited, it may result in fewer threads an SM can hold, and thus affect the selection of thread-block size. *Finally*, the many limits in GPU hardware add further complexity. In GeForce 8800 GT, for instance, the maximum number of threads per block is 512, the maximum number of threads per SM is 768, the maximum number of blocks per SM is 8, and at each time, all the threads assigned to an SM must use no more than 16 KB shared memory and 8192 registers in total. These constraints plus the unpredictable effects of optimizations make it extremely difficult to build an accurate analytical model for GPU optimization.

An alternative strategy for determining the best optimizations is through empirical search, whereby the optimizer searches for the best optimization parameters by running the GPU application many times, each time with different optimizations applied. Three obstacles must be removed before this solution becomes practical. First, a compiler is needed for abstracting out the optimization space and transforming the program accordingly. Second, effective space prunes are necessary for the search efficiency, especially when the optimization space is large. Finally, the optimizer must be able to handle the influence of program inputs. Our study (Section 5.5) shows that the best values of optimization parameters of some GPU programs are different for different inputs. For example, an optimization suitable for one input to a reduction program degrades the performance of the program on another input by as much as 640%. For such programs, it is desirable to detect the input-sensitivity and make the optimization cross-input adaptive.

5.4 Adaptive Optimization Framework

G-ADAPT is our solution to the challenges in GPU program optimization. It is a cross-input adaptive framework, unifying source-to-source compilation, performance modeling,

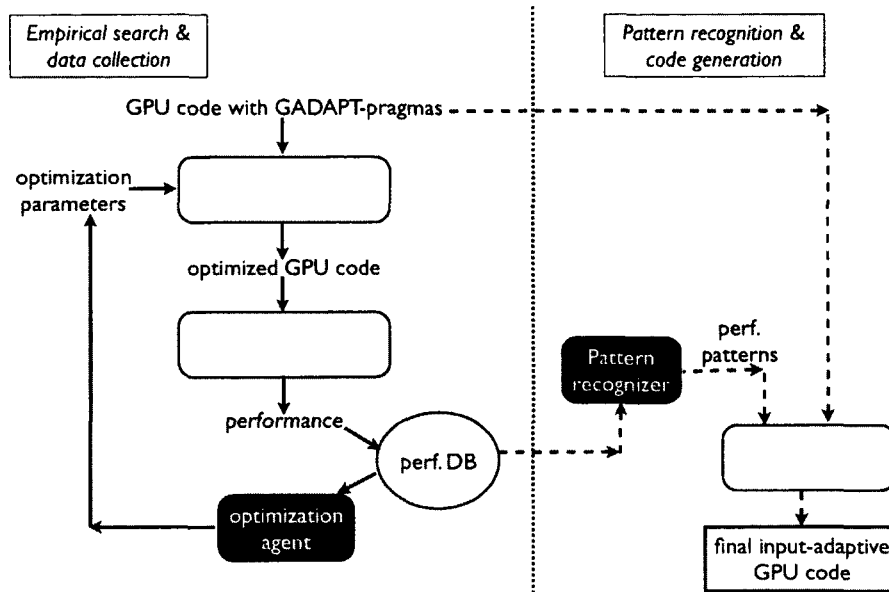


Figure 5.1: G-ADAPT: An adaptive optimization framework for GPU programs.

and pattern recognition. This section first gives an overview of the framework, and then elaborates on every component in the framework.

5.4.1 Overview

Figure 5.1 shows the structure of *G-ADAPT*. Its two parts separated by the dot vertical line correspond to two stages of the optimization. The task of the first stage, shown as the left part in Figure 5.1, is to conduct a series of empirical search in the optimization space of the given GPU program. During the search, a set of performance data, along with the program input features, are stored into a database. After the first stage finishes, the second stage, shown as the the right part of Figure 5.1, uses the performance database to recognize the relation between program inputs and the corresponding suitable optimization decisions. *G-ADAPT* then transforms the original GPU code into a program that is able to automatically adapt to an arbitrary input.

The first part uses empirical search to overcome the difficulty in modeling GPU program performance; the second part addresses the input-sensitivity issue by recognizing the

influence of inputs and making GPU program adaptive.

5.4.2 Stage 1: Heuristic-Based Empirical Search and Data Collection

The first stage is an iterative process. The inputs to the process include a given GPU application (with some pragmas inserted) with a set of typical inputs.

In the iterative process, the adaptive framework, for each of the given inputs to the GPU application, automatically searches for the best values of optimization parameters that can maximize the performance of the application. The process results in a performance database, consisting of a set of $\langle \text{input}, \text{best parameter values} \rangle$ tuples.

Three components are involved in this iterative process. For a given input to the GPU program, in each iteration, a compiler produces a new version of the application, a calibrator then measures the performance of the program on the given input, and the measured result is used by an optimization agent to determine what version of the program the next iteration should try. When the system finds the best optimization values for that input, it stores the values into the performance database, and starts the iterations for another input.

Several issues need to be addressed to make the empirical search efficient and widely applicable. The issues include how to derive optimization space from the application, how to characterize program inputs, and how to prune the search space to accelerate the search. In the following, we describe how the 3 components in the first stage of G-ADAPT work together to address these issues.

5.4.2.1 Optimization Pragmas and G-ADAPT Compiler

We classify the optimization parameters in GPU applications into three categories, corresponding to three different optimization levels. In the first category are execution configurations of the program—that is, the number of threads per block and the number of thread blocks for the execution of each GPU kernel. The second category includes the parameters that determine how the compiler transforms the program code, such as loop unrolling levels and size of loop tiles. The third category includes other implementation-level or algorithmic

decisions, such as the selection of different algorithms for implementing a function. These parameters together constitute the space for the empirical search.

Different applications have different parameters to optimize; some parameters may be implicit in a program, and the ranges of some parameters may be difficult to be automatically determined because of aliases, pointers, and the entanglement among program data.

So even though compilers may automatically recognize some parameters in the first two categories, for automatic search to work generally, it is necessary to have a mechanism to easily expose all those kinds of parameters and their possible values for an arbitrary GPU application.

In this work, we employ a set of pragmas, named G-ADAPT pragmas, to support the synergy between programmers and compilers in revealing the optimization space. There are three types of pragmas. The first type is dedicated for the adjustment of scalar variable (or constant) values that control the execution configurations of the GPU application. The second type is for compiler optimizations. The third type is for implementation selection. The pragmas allow the inclusion of search hints, such as the important value ranges of a parameter and the suitable step size. For example, a pragma, “`#pragma erange 64,512,2`” above the statement “`#define BLKSZ 256`”, means that the search range for the value of BLKSZ is from 64 to 512 with exponential (the first “e” in “erange”) increases with base 2.

We develop a source-to-source compiler, named the G-ADAPT compiler, to construct and explore the optimization space. The G-ADAPT compiler is based on Cetus [35], a C compiler infrastructure developed by the group led by Eigenmann and Midkiff. With some extensions added to Cetus, the G-ADAPT compiler is able to support CUDA programs, the G-ADAPT pragmas, and a set of program transformations (e.g., redundant elimination, and various loop transformations.)

The G-ADAPT compiler has two-fold responsibilities. At the beginning of the empirical search, the compiler recognizes the optimization space through data flow analysis, loop analysis, and analysis on the pragmas in the GPU application. In each iteration of the

empirical search, the compiler uses one set of parameter values in the search space to transform the application and produces one version of the application.

5.4.2.2 Performance Calibrator and Optimization Agent

The performance calibrator invokes the CUDA compiler, *NVCC*, to produce an executable from the GPU program generated by the G-ADAPT compiler. It then runs the executable (on the current input) to measure the running time. After the run, it computes the occupancy of the executable on the GPU. The occupancy reflects the degree to which the executable exerts the computing power of the GPU. A higher occupancy is often desirable, but does not necessarily suggest higher performance. The occupancy calculation is based on the occupancy calculating spreadsheet [1] provided by NVIDIA. Besides hardware information, the calculation requires the information on the size of shared memory allocated in each thread, the number of registers used by each thread, and the thread block size. The calibrator obtains the information from the “.cubin” files of the GPU program and the execution of the executable ¹.

The calibrator then stores the parameter values, along with the running time and occupancy, into the performance database. It checks whether the termination conditions (explained next) for the search on the current input have been reached; if so, it stores the input, along with the best parameter values that have been found, into the performance database.

The responsibility of the optimization agent is to determine which point in the optimization space should be explored in the next iteration of the search process. The size of the optimization space can be very large. For K independent parameters, with D_i denoting the number of possible values of the i th parameter, the optimization space is as large as $\prod_{i=1}^K D_i$. It implies that for an application with many loops and implementation options, the space may become too large for the framework to enumerate all the points. The optimization agent uses hill climbing to accelerate the search. Let K be the number

¹The “.cubin” files are generated by *NVCC* with the usage of registers and shared memory per thread block exposed.

of parameters. The search starts with all the parameters having their minimum values. In each of the next K iterations, it increases one parameter by a step and keeps the others unchanged. After iteration $(K + 1)$, it finds the best of the K parameter vectors that are just tried, and use it as the base for the next K iterations. This process continues. When one parameter reaches the maximum, it stops increasing. When all parameters reach their maximum values, the search stops.

This hill climbing search differs from the model-based prune proposed by Ryoo et al. [51]. Their approach is applicable when the program performance is not bounded by memory bandwidth; the method has shown more significant prune rate than our approach does. On the other hand, the hill climbing search is more generally applicable, making no assumptions on the GPU application.

5.4.3 Stage 2: Pattern Recognition and Cross-Input Adaptation

After the first stage, the performance database contains a number of $\langle \text{input}, \text{best parameter values} \rangle$ tuples, from which, the pattern recognizer learns the relation between program inputs and the optimization parameters. A number of statistical learning techniques can be used in the learning process. In this work, we select Regression Trees [27] for its simplicity and good interpretability. Regression Trees is a divide-and-conquer learning approach. It divides the input space into local regions with each region having a regular pattern. In the resulting tree, every non-leaf node contains a question on the input features, and every leaf node corresponds to a region in the input space. The question contained in a non-leaf node is automatically selected in the light of entropy reduction, defined as the increase of the purity of the data set after the data are split by that question. We then apply Least Mean Squares (LMS) to the data that fall into each leaf node to produce the final predictive models.

To capitalize on the learned patterns, we need to integrate them into the GPU application. If there were just-in-time compiler (JIT) support, the integration could happen during runtime implicitly: The JIT compiles the program functions using the parameters

predicted as the best for the program input. Without JIT, the integration can occur either through a linker, which links the appropriate versions of object files into an executable before every execution of the application, or an execution wrapper, which every time selects the appropriate version of executables to run. In our experiments, we use the wrapper solution because it has no linking overhead, and the programs in our experiments need only few versions of executables. The G-ADAPT compiler, along with the CUDA compiler, produces one executable for each parameter vector that is considered as the best for some training inputs in the performance database. When the application is launched with an arbitrary input, the version selector in the wrapper uses the constructed regression trees to quickly determine the right executable based on the input and then runs the program.

5.5 Evaluation

We use seven benchmarks to test the effectiveness of the optimization framework, as listed in Table 5.1. Most of the programs are from NVIDIA SDK [1]. The program, *mvMul*, is a matrix vector multiplication program from Fujimoto [23]. It is an efficient implementation, outperforming the NVIDIA CUBLAS [1] version significantly, thanks to its adoption of a new algorithm along with an effective use of texture memory [23].

We emphasize that the programs we use have all been manually tuned by the developers. The *reduction* program, for instance, has gone through seven optimizations, respectively on the algorithm, locality, branch divergence, loop unrolling and so on. NVIDIA has used it as a typical example to demonstrate manual optimizations on GPU programs. The sequence of optimizations have accelerated the program by as much as a factor of 30 [26].

The third column of the table shows the number of different inputs we have used for each benchmark. We create those inputs based on our understanding to the applications, with an attempt to cover a wide range of the input space.

The type of GPU we use is NVIDIA GeForce 8800 GT. It contains 512 MB global memory, 14 multiprocessors, 112 cores, with clock rates set at 1.51 GHz. Each multipro-

Table 5.1: Benchmarks for input adaptivity experiments on GPU

Benchmark	Description	Num of Inputs	Prediction Accuracy	Training iterations	Training time (s)
convolution	convolution filter of a 2D signal	10	100%	200	2825
matrixMul	dense matrix multiplication	9	100%	196	2539
mvMul	dense matrix-vector multiplication	15	93.3%	124	124
reduction	sum of array	15	80%	75	29
scalarProd	scalar products of vector pairs	7	100%	93	237
transpose	matrix transpose	18	100%	54	1639
transpose-co	matrix transpose with coalescing memory references	18	100%	54	631

cessor has 16 KB shared memory and 8192 registers. Every GPU co-runs with 2 Intel Xeon processors (3.6 GHz) on a machine with SUSE Linux 2.6.22 installed.

Before presenting the detailed results on each benchmark, we briefly summarize the results. The best configurations of three out of the seven programs change with their inputs. For all the programs, the G-ADAPT is able to learn the relation between inputs and optimization parameters, producing over 93% prediction accuracy (except 80% for one program) for the best optimization decisions. The prediction yields several times of speedup compared the running times of the original programs. In the following subsections, we present the results of the input-sensitive programs first, followed by the results of other programs.

5.5.1 Matrix-Vector Multiplication

The program, *mvMul*, computes the product between a dense matrix and a vector. The parameters of this program include the size of a thread block, and the loop unrolling factors in the kernel function. Figure 5.2 shows the performance of the program on two example inputs when different configurations are used. The different parameter values cause up to 2.5 times performance difference. The block size has more significant influence than the unrolling levels. Moreover, the results clearly show the influence of program inputs on the

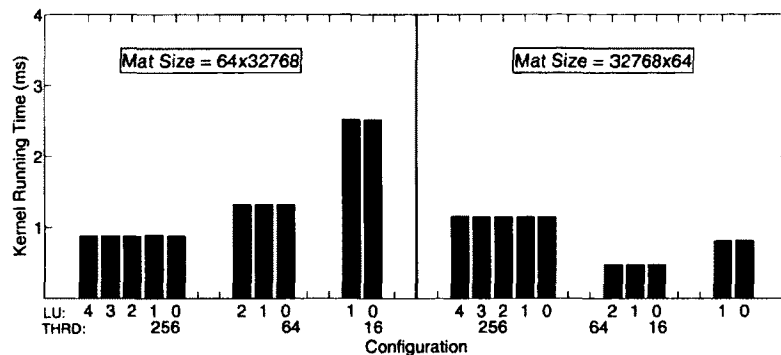


Figure 5.2: Performance of matrix-vector multiplication on two inputs when different optimization decisions are used. LU: loop unrolling levels; THRD: number of threads per block. The maximum unrolling level can only be $\sqrt{THRD}/4$.

optimal parameter values. The best block size for the first input turns out to be the worst for the second input, causing 2.4 times slowdown than its best run. One of the reasons for the negligible effect of loop unrolling is that there is little room for adjustment: the innermost loop can have iterations of at most a quarter of the width of a thread block.

Figure 5.3 (a) reports the best block size for each of the 15 inputs. The block size used in the original program is 256, which works the best for the 4 inputs on the left. For the other inputs, the best block size is 64. Figure 5.3 (b) plots the speedups of the program when it uses the optimizations predicted by the G-ADAPT framework. The baseline is the running times of the original program. The trend is that as the height of the input matrix becomes larger than its width, the speedup becomes larger. The reason why large blocks work poorly for thin matrices is that each time, a block is in charge of a group of rows, and in thin matrices, each thread has little work to do and thus results in low occupancy on GPU processors. This benchmark demonstrates that the shape of the input matrix is critical for the optimization decisions.

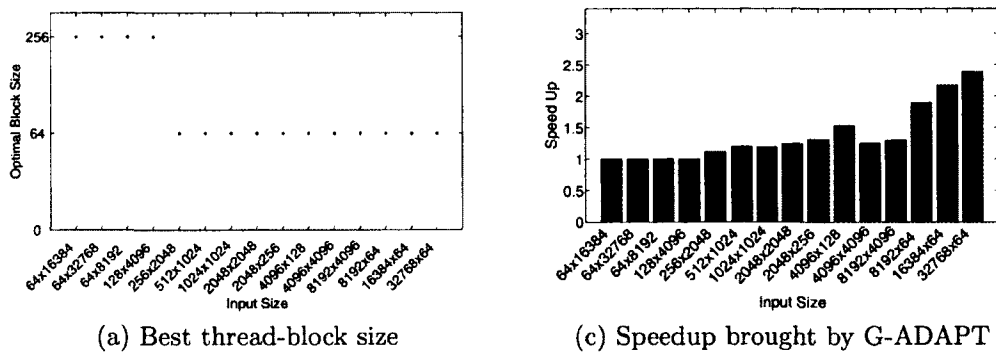


Figure 5.3: The best values of the optimization parameters of *mvMul* are input-sensitive. G-ADAPT addresses the influence and produces significant speedup compared to the original program.

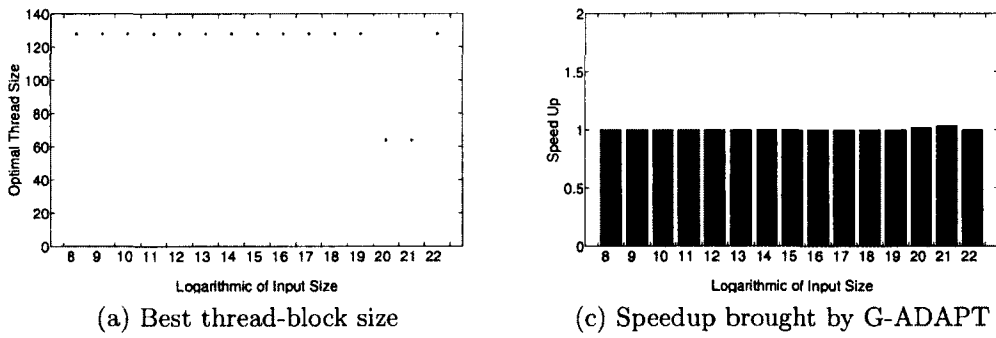


Figure 5.4: Experimental results on *reduction*.

5.5.2 Parallel Reduction

The program, *reduction*, performs sum operations on an array of integers. It represents one kind of common computation in parallel computing, reducing a series of values into a single value. Given that many optimizations (e.g., loop unrolling) have been manually applied in the development of the original program, our experiment concentrates on a single parameter, the number of threads per block.

The default setting is 128 threads per block. That setting turns out to be the best for most inputs, except two inputs whose array sizes are 2^{19} and 2^{20} , in which case, the best block size is 64. Even on these two inputs, the default setting works virtually similar to the optimal, with only 3% performance difference.

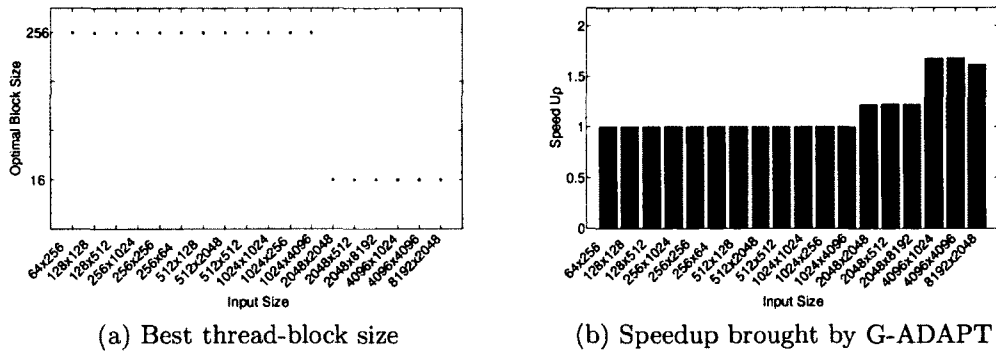


Figure 5.5: Experimental results on *transpose*.

5.5.3 Matrix Transpose

There are two versions of matrix transpose in the NVIDIA SDK. One uses memory coalescing and the other one does not; we denote them as *transpose-co* and *transpose* respectively. In both versions, the kernel function contains no loops, and the key optimization parameter is the block size. Figure 5.5 shows the results of *transpose*. For matrices of medium sizes, the best block size is 256, the same as the default setting in the original program. Whereas, the best size becomes 16 when the matrix size increases to over 4 million elements. The speedup becomes more significant as the matrix become larger.

In contrast, the coalesced version, *transpose-co*, is not input-sensitive. The best block size is always 256. This version differs from *transpose* mainly in memory accesses. In the kernel function of *mtco*, the references to the global memory are staged. The data are first brought into shared memory in a coalesced manner before the computation. Furthermore, the array is padded to reduce bank conflicts in the shared memory. The changes in memory reference patterns remove the input-sensitivity. When the block size is 16, the program achieves 100% occupancy on the multiprocessors, and thus exhibits the best performance.

5.5.4 Other Benchmarks and Overall Results

The best values of the parameters in the other 3 benchmarks, *matMulGPU*, *convolution*, *scalarProd*, show no sensitivity to their inputs. Besides the parameters for loop optimiza-

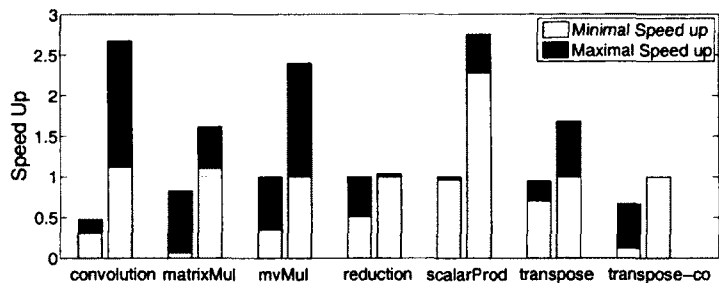


Figure 5.6: The ranges of speedup brought by different optimization decisions. For each program, the left bar shows the range of speedup (less than 1 means slowdown) if the worst decision is taken. The right bar shows the range of speedup when the G-ADAPT’s prediction is used.

tions, the program *matMulGPU* has a parameter controlling the size of thread blocks, the program *convolution* has 3 parameters controlling the tile size and the number of columns, and the program *scalarProd* has 2 parameters controlling the dimensions of the grid and the dimensions of a thread block. The G-ADAPT system successfully finds the best parameter values for all the 3 programs.

We apply the predictions of G-ADAPT to these programs to measure the effectiveness in performance improvement. The prediction is based on leave-one-out cross validation [27], which is a typical practice in statistical learning to estimate the error of a predictive model in real uses. For each input, we use all the other inputs as training inputs to build regression trees, and then apply the trees to the left-out input to predict the corresponding best optimization decisions. The average prediction accuracies are shown in the fourth column in Table 5.1. For input-insensitive programs, the prediction is simple. For the input-sensitive programs, the prediction accuracy is 80% for *reduction*, 93.3% for *mvMul*, and 100% for *transpose*. These results demonstrate the effectiveness of the Regression Trees method in modeling the relation between inputs and optimization decisions.

On different inputs to an application, the G-ADAPT yields different speedups. Figure 5.6 summarizes the ranges of speedup brought by G-ADAPT on the 7 GPU programs. The baseline is the running times of the original GPU programs. For each program, the

left bar in a benchmark corresponds to the worst configuration encountered in the explored optimization space, which reflects the risk of a careless configuration or transformation. The right bar shows the effectiveness of G-ADAPT. Among all programs, only the default settings in *transpose-co* and *reduction* happen to be (almost) the same as the one G-ADAPT finds. The 1.5 to 2.8 times of speedup on other programs demonstrate the effectiveness of input-adaptive optimizations enabled by G-ADAPT.

5.6 Discussions

In this section, we first present the training overhead of G-ADAPT and then discuss some complexities in applying G-ADAPT for large applications.

The right-most two columns in Table 5.1 reveal the training overhead of G-ADAPT on the seven benchmarks. The total numbers of iterations range from 54 to 200, and the total training time spans from 29 seconds to 47 minutes. The time is determined by the number of training inputs, the dimensions of the search space, and the size of the inputs. The program, *convolution*, happens to run for a long time on some of its training inputs, resulting in the longest training time.

It is worth noting that one complexity, input characterization, happens to be simple in our experiments. Input characterization is to determine the important features of program inputs. In our experiments, the inputs to the programs are just several numbers, indicating the sizes of the input signal, matrix, array, or vector, which naturally capture the important characteristics of the input data sets. However, for large complex GPU applications, the input characterization may need special treatment. One option is to develop some input characterization procedures and link them with G-ADAPT. A recent study [40] proposes an extensible input characterization language, XICL, to ease the efforts. Detailed studies remain to be our future work.

5.7 Related Work

The studies closest to this work are the recent explorations by Ryoo et al. [51], and Baskaran et al. [6]. Ryoo and his colleagues have defined *efficiency* and *utilization* models for GPU computing, and demonstrated the effectiveness of the models in pruning of the optimization space. Our study complements their technique in that the influence from program inputs is a dimension omitted in their work. Furthermore, the previous work conducts transformations manually, whereas, we develop a compiler framework with optimization pragmas for automatic transformations. The prune method in our tool complements the previous models in that it relaxes some assumptions made by previous work, such as the memory bandwidth is not the bottleneck on performance. On the other hand, the previous models may work well in the cases when the assumptions hold.

In the study by Baskaran et al. [6], the authors focus on the optimization of affine loops in GPU applications. They develop an approach to improving global memory accesses and use model-driven empirical search to determine optimal parameters for loop unrolling and tiling. Our work is complementary to their technique on two aspects. First, our optimizations are input adaptive, whereas, the influence of program inputs is a missing factor in the previous study. Second, our tool can be applied to not only optimization of affine loops, but also other factors that affect the performance of GPU applications, such as the size of thread block size and implementation-level decisions. On the other hand, the transformations developed in the previous work can strengthen the effectiveness of our tool. An integration of them into the tool may be worthwhile.

On traditional CPU architecture, there has been many studies on empirical-search based optimizations. Many of the explorations are for the development of efficient numerical libraries or kernels, such as ATLAS [70], PHiPAC [9], SPARSITY [29], SPIRAL [46], FFTW [22], STAPL [64]. Our work is enlightened by those explorations, but focuses on a single-chip massively parallel architecture, on which, the optimizations dramatically differ from those on the previous CPU architecture. Furthermore, the targets of this work are gen-

eral GPU applications, rather than a certain set of kernels. The variety in the applications further complicates input characterization and the construction of cross-input predictive models.

The adaptation to different program inputs in this work shares some common theme with code specialization, such as procedure cloning [15], the incremental run-time specialization [41], the specialization of libraries in Telescoping Languages [33]. In addition, dynamic optimizations [4,18,40,69] may tailor a program to their inputs by runtime code generation.

5.8 Conclusion

This paper reports our exploration of the influence of program inputs on GPU program optimizations. It shows that for some GPU applications, their best optimizations are different for different inputs. It presents a compiler-based adaptive framework, G-ADAPT, which is able to extract optimization space from program code, and automatically search for the best optimizations for an GPU application on different inputs. With the use of Regression Trees, G-ADAPT produces cross-input predictive models from the search results. The models can predict the best optimizations from the input given to the GPU application, and thus enable cross-input adaptive optimizations. Experiments show significant performance improvement generated by the optimizations, demonstrating the promise of the framework as an automatic tool for resolving the productivity bottleneck in the development of efficient GPU programs.

Chapter 6

Conclusion

In conclusion, my PhD research focuses on revealing and exploiting the implications of emerging hardware features on the development, compilation, and execution of software. Enhancing the match between software executions and hardware features is key to computing efficiency (in terms of both performance and energy). The match is a continuously evolving and challenging problem. I have concentrated on the development of programming system support for exploiting two key features of modern hardware development: the massive parallelism of emerging computational accelerators such as Graphic Processing Units (GPU), and the non-uniformity of cache sharing in modern multicore processors. They are respectively driven by the important role of accelerators in today's general-purpose computing and the ultimate importance of memory performance. More specifically, my research has initiated extensive research efforts on optimizing control flow and memory reference at the program level in order to take advantage of these two key hardware features.

Conditional branches cause divergences in program control flows, which may result in serious performance issues on massively data-parallel GPU architectures with SIMD parallelism. On such an architecture, control divergence may force computing units to stay idle for a substantial time, throttling system throughput by orders of magnitude. My research has attacked this problem by proposing two types of program transformations. The first is thread relocation, which reorganizes logical threads by reassigning their thread IDs to

minimize the divergences among their control flows. The second is data relocation, which switches the tasks of threads by relocating data in memory without changing thread IDs and data reference indices. These two optimizations provide fundamental support for swapping jobs among threads so that the control flow paths of threads converge within every SIMD thread group. Compared to previously proposed hardware extensions, my software approaches are readily deployable and address dynamic program behavior by adapting to different program inputs and phases. To the best of our knowledge, this work is the first in a series of research efforts that addresses control flow divergence elimination problem on GPUs through software support.

My PhD research in memory performance has concentrated on two aspects of the problem: the influence of non-uniform sharing on multithreading applications, and the optimization of irregular memory references on GPUs. In shared cache multicore chips, interaction among threads are complicated due to the interplay of cache contention and synergistic prefetching. I conducted the first extensive investigation of the influence of shared cache on contemporary multi-threaded applications, with important factors systematically examined ranging from architecture level to program level. My study reveals the surprisingly insignificant influences on the performance of the application by shared cache. I identified the underlying reason that current program development is oblivious to the non-uniformity in cache sharing topology. I proposed a novel data locality optimization paradigm to match with the non-uniformity of cache sharing, which opens up many new opportunities for addressing the memory performance issues in multicore and manycore systems. The efficiency of GPU accelerator is sensitive to irregular memory references, which refer to the memory references whose access patterns remain unknown until execution time (e.g., $A[P[i]]$). Experiments have shown great performance gains when these irregularities are removed. But it remains an open question how to achieve those gains through software approaches. My work presented a systematic exploration on this problem. I discovered that root causes of irregular memory reference problem are similar to that of the control flow problem, while in a more general and complex form. They both may arise from the undesirable thread-data

mapping while irregular memory reference may also come from the original data layout. I explored the inherent properties of both irregularities, including their interactions, their relations with program data and threads, the computational complexities in removing them, and heuristics-based algorithms for their removal through data reordering, job swapping, and hybrid transformations. I developed a framework, named G—Streamline, as a unified software solution to dynamic irregularities in GPU computing. It treats both types of irregularities at the same time in a holistic fashion, maximizing the whole-program performance by resolving conflicts among optimizations. Its optimization overhead is largely transparent to GPU kernel executions, without jeopardizing the basic efficiency of the GPU application.

References

- [1] NVIDIA CUDA. <http://www.nvidia.com/cuda>.
- [2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, August 2006.
- [3] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, 2001.
- [4] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of Java. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 111–129, 2002.
- [5] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report 103863, NASA, July 1993.
- [6] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for GPGPUs. In *ICS*, 2008.
- [7] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 47–56, 2008.

- [8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.
- [9] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *Proceedings of the ACM International Conference on Supercomputing*, pages 340–347, 1997.
- [10] S. Browne, C. Deane, G. Ho, and P. Mucci. PAPI: A portable interface to hardware performance counters. In *Proceedings of Department of Defense HPCMP Users Group Conference*, 1999.
- [11] S. Carrillo, J. Siegel, and X. Li. A control-structure splitting optimization for gpgpu. In *CF*, 2009.
- [12] J. Chang and G. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the 21st annual international conference on Supercomputing*, pages 242–252, 2007.
- [13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.
- [14] T. M. Chilimbi and R. Shaham. Cache-conscious coallocation of hot data streams. In *PLDI*, 2006.
- [15] K. D. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *Computer Languages*, pages 96–105, 1992.
- [16] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the SIG-PLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

- [17] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *JPDC*, 2004.
- [18] P. Diniz and M. Rinard. Dynamic feedback: an effective technique for adaptive computing. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 71–84, Las Vegas, May 1997.
- [19] Y. Dotsenko, N. K. Govindaraju, P. Sloan, C. Boyd, and J. Manferdelli. Fast scan algorithms on graphics processors. In *ICS'08: Proceedings of the 22nd Annual International Conference on Supercomputing*, pages 205–213, 2008.
- [20] Ali El-Moursy, R. Garg, D. H. Albonesi, and S. Dwarkadas. Compatible phase co-scheduling on a cmp of multi-threaded processors. In *Proceedings of the International Parallel and Distribute Processing Symposium (IPDPS)*, 2006.
- [21] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 25–38, 2007.
- [22] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.
- [23] N. Fujimoto. Faster matrix-vector multiplication on GeForce 8800GTX. In *Proceedings of the Workshop on Large-Scale Parallel Processing (co-located with IPDPS)*, pages 1–8, 2008.
- [24] W. Fung, I. Sham, G. Yuan, and T. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *MICRO*, 2007.
- [25] H. Han and C.-W. Tseng. Exploiting locality for irregular scientific codes. *IEEE Transactions on Parallel Distributed Systems*, 17(7):606–618, 2006.
- [26] M. Harris. High performance computing with CUDA. In *Tutorial in IEEE SuperComputing*, 2007.

- [27] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning*. Springer, 2001.
- [28] D. S. Hochbaum. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1995.
- [29] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, 2004.
- [30] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 220–229, October 2008.
- [31] Y. Jiang, K. Tian, and X. Shen. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. In *Proceedings of The International Conference on High Performance Embedded Architectures and Compilation (HiPEAC)*, pages 201–215, 2010.
- [32] Yunlian Jiang, Eddy Z. Zhang, Xipeng Shen, Yaoqing Gao, and Roch Archambault. Array regrouping on cmp with non-uniform cache sharing. In *Proceedings of the 23rd international conference on Languages and compilers for parallel computing, LCPC'10*, pages 92–105, Berlin, Heidelberg, 2011. Springer-Verlag.
- [33] K. Kennedy, B. Broom, A. Chauhan, R. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey. Telescoping languages: A system for automatic generation of domain languages. *Proceedings of the IEEE*, 93(2):387–408, 2005.
- [34] R. Kumar and D. Tullsen. Compiling for instruction cache performance on a multi-threaded architecture. In *Proceedings of the International Symposium on Microarchitecture*, pages 419–429, 2002.

- [35] S. Lee, T. Johnson, and R. Eigenmann. Cetus - an extensible compiler infrastructure for source-to-source transformation. In *Proceedings of the 16th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 539–553, 2003.
- [36] S. Lee, S. Min, and R. Eigenmann. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In *PPoPP*, 2009.
- [37] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik. Locality and loop scheduling on NUMA multiprocessors. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 140–147, 1993.
- [38] C. Liao, Z. Liu, L. Huang, and B. Chapman. Evaluating OpenMP on chip multithreading platforms. In *Proceedings of International Workshop on OpenMP*, 2005.
- [39] Y. Liu, E. Z. Zhang, and X. Shen. A cross-input adaptive framework for gpu programs optimization. In *Proceedings of International Parallel and Distribute Processing Symposium (IPDPS)*, pages 1–10, 2009.
- [40] F. Mao and X. Shen. Cross-input learning and discriminative prediction in evolvable virtual machine. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 92–101, 2009.
- [41] R. Marlet, C. Consel, and P. Boinot. Efficient incremental run-time specialization for free. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 281–292, Atlanta, GA, May 1999.
- [42] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *ISCA*, 2010.
- [43] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *ACM Queue*, pages 40–53, March/April 2008.

- [44] D. Nikolopoulos. Code and data transformations for improving shared cache performance on smt processors. In *Proceedings of the International Symposium on High Performance Computing*, pages 54–69, 2003.
- [45] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka. Bandwidth intensive 3-D FFT kernel for GPUs using CUDA. In *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–11, 2008.
- [46] M. Puschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo. SPIRAL: code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.
- [47] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the International Symposium on Microarchitecture*, pages 423–432, 2006.
- [48] N. Rafique, W. Lim, and M. Thottethodi. Architectural support for operating system-driven CMP cache management. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 2–12, 2006.
- [49] G. Rudy, C. Chen, M. Hall, M. Khan, and J. Chame. Using a programming language interface to describe gpgpu optimization and code generation. 2010.
- [50] S. Ryoo, C. I. Rodrigues, S. S. Bagsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP*, 2008.
- [51] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Bagsorkhi, S. Ueng, J. A. Stratton, and W. W. Hwu. Program optimization space pruning for a multithreaded GPU. In *CGO'08: Proceedings of the Sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 195–204, 2008.

- [52] S. Sarkar and D. Tullsen. Compiler techniques for reducing data cache miss rate on a multithreaded architecture. In *Proceedings of The HiPEAC International Conference on High Performance Embedded Architectures and Compilation*, pages 353–368, 2008.
- [53] A. Settle, J. L. Kihm, A. Janiszewski, and D. A. Connors. Architectural support for enhanced SMT job scheduling. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 63–73, 2004.
- [54] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 165–176, 2004.
- [55] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, 2002.
- [56] Mark Silberstein, Assaf Schuster, Dan Geiger, Anjul Patney, and John D. Owens. Efficient computation of sum-products on GPUs through software-managed cache. In *Proceedings of the 22nd ACM International Conference on Supercomputing*, pages 309–318, June 2008.
- [57] A. Snaveley and D.M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 66–76, 2000.
- [58] A. Snaveley, D.M. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 66–76, 2002.
- [59] G.E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 117–128, 2002.

- [60] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. *SIGOPS Oper. Syst. Rev.*, 41(3):47–58, 2007.
- [61] G. Tan, Z. Guo, M. Chen, and D. Meng. Single-particle 3d reconstruction from cryo-electron microscopy images on gpu. In *ICS*, 2009.
- [62] D. Tarjan, J. Meng, and K. Skadron. Increasing memory miss tolerance for simd cores. In *SC*, 2009.
- [63] NVIDIA Tesla Bio Workbench. http://www.nvidia.com/object/tesla_bio_workbench.html.
- [64] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 277–288, 2005.
- [65] K. Tian, Y. Jiang, and X. Shen. A study on optimally co-scheduling jobs of different lengths on chip multiprocessors. In *Proceedings of ACM Computing Frontiers*, pages 41–50, 2009.
- [66] N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pages 26–35, 2003.
- [67] S. Ueng, S. Bagsorkhi, M. Lathara, and W. Hwu. Cuda-lite: Reducing gpu programming complexity. In *LCPC*, 2008.
- [68] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.
- [69] M. Voss and R. Eigenmann. High-level adaptive program optimization with ADAPT. In *Proceedings of ACM Symposium on Principles and Practice of Parallel Programming*, pages 93–102, Snowbird, Utah, June 2001.

- [70] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.
- [71] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture*, pages 24–36, 1995.
- [72] Bo Wu, Eddy Z. Zhang, and Xipeng Shen. Enhancing data locality for dynamic simulations through asynchronous data transformations and adaptive control. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, 2011.
- [73] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A gpgpu compiler for memory optimization and parallelism management. In *PLDI*, 2010.
- [74] E. Z. Zhang, Y. Jiang, Z. Guo, and X. Shen. Streamlining gpu applications on the fly. In *ICS*, 2010.
- [75] E. Z. Zhang, Y. Jiang, and X. Shen. Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 203–212, 2010.
- [76] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 369–380, New York, NY, USA, 2011. ACM.
- [77] Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. The significance of cmp cache sharing on contemporary multithreaded applications. *IEEE Transactions on Parallel and Distributed Systems*, 23:367–374, 2012.

- [78] Y. Zhao. Lattice boltzmann based pde solver on the gpu. *The Visual Computer*, (5):323–333, 2008.

VITA

Zheng Zhang is originally from China, where she received her Bachelor of Science degree in Electronic Engineering from Shanghai Jiao Tong University in 2004. She received her Master of Science degree in Computer Science from the College and William and Mary in 2007. She began her work on Doctor of Philosophy from Aug 2007. Her research interests lie generally in the area of compilers and programming systems, with a focus on revealing and exploiting the implications of emerging hardware features on the development, compilation, and execution of software. This dissertation was defended in May 31, 2012.