2014

Matching non-uniformity for program optimizations on heterogeneous many-core systems

Bo Wu
College of William & Mary - Arts & Sciences

Matching Non-Uniformity for Program Optimizations on Heterogeneous Many-Core Systems

Bo Wu

Puyang, Henan, China

Master of Science, Central South University, 2008
Bachelor of Science, Central South University, 2005

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, 2014

APPROVAL PAGE

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

Doctor of Philosophy

Bo Wu

Approved by the Committee, July, 2014

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

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

Associate Professor Haining Wang, Computer Science
The College of William and Mary

Senior computer scientist Jie Chen
Jefferson Lab

Professor Keshav Pingali, Computer Science
University of Texas, Austin

ABSTRACT

As computing enters an era of heterogeneity and massive parallelism, it exhibits a distinct feature: the deepening non-uniform relations among the computing elements in both hardware and software. Besides traditional non-uniform memory accesses, much deeper non-uniformity shows in a processor, runtime, and application, exemplified by the asymmetric cache sharing, memory coalescing, and thread divergences on multicore and many-core processors. Being oblivious to the non-uniformity, current applications fail to tap into the full potential of modern computing devices.

My research presents a systematic exploration into the emerging property. It examines the existence of such a property in modern computing, its influence on computing efficiency, and the challenges for establishing a non-uniformity–aware paradigm. I propose several techniques to translate the property into efficiency, including data reorganization to eliminate non-coalesced accesses, asynchronous data transformations for locality enhancement and a controllable scheduling for exploiting non-uniformity among thread blocks. The experiments show much promise of these techniques in maximizing computing throughput, especially for programs with complex data access patterns.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

First of all, I would like to thank my research advisor, Professor Xipeng Shen, for his top-notch guidance throughout my Ph.D. study. An extremely passionate researcher, Xipeng showed me the beauty of scientific research, the correct attitudes towards details and the dedication to make things done. My research journey with him was full of adventures and excitement, through which I developed myself to be a better researcher and also a better person.

I also owe thanks to my committee members, Professor Weizhen Mao, Haining Wang, Keshav Pingali and Dr. Jie Chen. Their kind advice made me more confident; their comments and suggestions helped shape this research.

I thank Yaoqing Gao, Raul Silvera, Graham Yiu, Dong Li, Jeffrey Vetter, Dong Ping Zhang and Nuwan Jayasena for their help during my internships. Those experiences taught me a lot of real-world practice.

I was lucky to be part of a great research group. The paper discussions with Yunlian jiang, Zheng Zhang, Kai Tian, Zhijia Zhao, Mingzhou Zhou and Ziyu Guo tremendously helped me develop my knowledge of the field. I also thank Guoyang Chen, Tao Wang, Qi Zhu, Yue Zhao, Weilin Wang and Yufei Ding for creating an inspiring spirit in this group.

Very importantly, I thank Zhijia Zhao, Jianhua Sun, Yunhan Long, Jianing Zhao, Xing Gao, Zijiang Hao, Yubao Zhang, Yue Li, Gang Zhou, Shanhe Yi and many others for playing basketball regularly with me. It was not only helpful for me to keep physically healthy during the tough Ph.D. journey, but also provided me with the relaxing moments to keep me a happy person.

I thank my family for always being there whenever I need help. My parents and parents in-law gave me substantial support, without which I could not focus on research. My little sister made so many great changes in her college life, which cheered me up to change myself to be a better person. Finally and most importantly, my wife and daughter deserve my greatest gratitude. Their love and support make what I do meaningful.

This Ph.D. is dedicated to my wife, my daughter and my parents for their love and support

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 The Problem

Heterogeneous systems leverage Graphics Processing Unites (GPUs) to speedup both compute-intensive and memory-intensive applications in many domains, including high-performance simulation [7,74], database systems [8,30], Big Data [64, 94] and so on. Being able to execute a large number of threads simultaneously on hundreds or even thousands of cores, GPUs provide tremendous peak computing power (e.g., 1.31 TFLOPS of double precision floating point performance on NVIDIA K20 GPUs) and great power efficiency (e.g., 9 times more energy efficient for some applications over CPU [42]). Previous studies [10, 80, 91] observed 1.16X to 431X speedups for various applications and workloads on GPUs compared to single-threaded CPU computations.

Off-chip memory access, however, is a bottleneck to fully exert the computing power of GPUs. The many cores issue a huge number of memory requests that the current memory system cannot handle efficiently. A recent study showed that for some applications GPU cores are idle for more than 90% of the execution time, waiting for data from main memory [47]. Due to the trend of designing simpler cores and integrating more cores on the same die, the gap between on-chip computing power and off-chip memory bandwidth is even widening. It is expected that

in future exascale supercomputers produced around 2020, the aggregate computing power will increase by hundreds of times, while the main memory bandwidth will only increase by around ten times [32]. The issue is further underscored by the advent of the Big Data processing era. Many data-intensive applications (e.g., dynamic simulation and graph processing) only do lightweight processing on each data element and hence do not significantly reuse the data in fast on-chip caches, further increasing the burden on the memory system.

Algorithm 1 *Molecular Dynamics Simulation*

```
1: Initialize atoms and neighbor_list
2: for iter = 0 to max_iterations − 1 do
3:   for i = 0 to num_atoms do
4:     atom_i = atoms[i];
5:     for j = 0 to num_neighbors − 1 do
6:       atom_j = atoms[neighbor_list[i][j]];
7:       atom_i = update_atom(atom_i, atom_j);
8:   if iter mod K == 0 then
9:     update(neighbor_list);
```

While memory optimization is the key to convert computing power to performance, it faces challenges posed by two trends in software and hardware. First, data-intensive applications usually have irregular memory access patterns that change during run time. For example, Algorithm 1 shows the pseudo-code of molecular dynamics simulation, which simulates the interactions among a large number of atoms. In the initialization phase (line 1), the code initializes the $atoms$ array, which stores various types of information of the atoms, and a $neighbor\_list$ array, which records the neighborhood relations among the atoms. Note that we need the $neighbor\_list$ array because of the atoms' non-uniform interactions: Each atom is modeled to only interact with its several neighbor atoms. Two atoms are neighbors if the distance between them is smaller than some threshold. The computation phase (line 3-7), updates every atom according to the interaction between itself and each of its neighbors. The accesses to the neighbor atoms (line 7) are through an index array $neighbor\_list$. Hence, the access pattern is de-

termined by the values in $neighbor\_list$ and can be rather irregular. Meanwhile, this pattern may change during run-time in that $neighbor\_list$ is updated every K iterations as shown by line 8 and 9.

The second trend is that the hardware shows deepened non-uniformity in both inter-core resource sharing and intra-core execution. The complex resource sharing complicates memory optimizations. For example, an NVIDIA Kepler GPU [71] contains multiple streaming multi-processors (named SMX), each having a private read-only data cache that can not be accessed by other multi-processors. But the L2 cache is uniformly shared by all multi-processors. Like traditional CPUs, GPU's cache efficiency also heavily depends on the regularity of memory accesses. Inside one multi-processor, the threads are organized into Single-Instruction-Multiple-Data (SIMD) groups, each of which executes in lockstep. Memory throughput significantly improves through memory coalescing if the threads in the same warp access close-by memory elements. Unfortunately, irregular memory accesses seriously limit the benefit of memory coalescing. On an NVIDIA GPU, the throughput degradation due to irregular memory accesses can be up to 32 times.

The non-uniformity in software and hardware leads to a mismatch that causes a double-digit performance degradation compared to peak performance [80, 99]. In this dissertation, we aim to bridge this gap through systematically exploring the influence of non-uniformity and designing a holistic non-uniformity-aware optimization framework.

1.2 Existing Approaches to Optimizing Memory Performance

Existing memory optimization studies for heterogeneous systems can be grouped into three categories: static approaches, dynamic approaches and hardware approaches. We briefly discuss their applicability and limitations.

Static Approaches: Static memory optimizations are usually integrated into compilers and try to infer memory access patterns through code analysis. Yang et al. [97] designed a source-to-source optimizing compiler, which considers multiple performance aspects, including vectorization, coalescing and data prefetching. For memory performance improvement, their main technique is to transform non-coalesced pattern—in which the threads in the same warp need to load more than one memory segment—into a coalesced one through using shared memory. Their access pattern detection can handle constant index, predefined index and loop index. However, in some applications like the one shown in Algorithm 1, the pattern depends on the values in some index array, which fails their detection algorithm and is hence ignored.

Verdoolaege et. al. [90] leverage polyhedral analysis to automatically parallelize affine loops and generate GPU code, with optimizations for memory coalescing and cache reuses. Similar to Yang's work, their technique's application is limited to code whose memory access pattern can be analyzed statically. Jia and others [44] observe the benefit of L1 cache bypassing for NVIDIA GPUs and proposed a compiler transformation to selectively enable bypassing for each global memory load instruction. They try to reduce the negative influence of the private data cache, rather than leverage the non-uniformity to improve data sharing.

Sung and others [86] focus on structured grid many-core applications. Since the memory access pattern is mostly determined by the source code, they propose compiler optimizations to transform data layout for better memory coalescing.

Dynamic Approaches: The idea of dynamically optimizing code whose memory access pattern is determined online dates back to the seminal work from Das et al. [25]. The work considers distributed irregular programs and transforms sequential loops into two constructs: the inspector and the executor. The inspector examines the access pattern and prepares optimized data (e.g., through prefetching) for the executor, which computes the output.

Chen and Kennedy [26] use the inspector-executor strategy for shared-memory programs—which are also the focus of this dissertation—and propose runtime analysis and data transformation to address dynamic memory access irregularity for CPU programs. Following their work, Han et al. [37] propose some more sophisticated data transformation techniques. The proposed techniques all happen on the critical path of the program: Only after they finish can the optimized code be executed. But the transformation overhead of some algorithms is non-trivial. For example, the recursive coordinate bisection algorithm may run as long as tens of invocations of the kernel function. Although lightweight algorithms, such as data packing, have much smaller overhead, their benefit is also significantly reduced. The overhead-benefit dilemma seriously limits the applicability of data transformation in real-world applications.

Zhang and others [99] propose the first runtime approach to address dynamic irregular memory references on GPU. They design a pipelined framework, in which the CPU transforms data for the GPU execution. The key insight is that irregularity can be easily removed through data duplication. However, space overhead seriously limits the applicability due to the capped device memory size and the transformation benefit due to the increased volume of data to be transferred

between CPU and GPU. The study lacks an in-depth study of the trade-off among transformation overhead, space cost and benefit.

Some studies use domain-specific knowledge to reorder data for better locality. Nasre and others [67, 68] optimize the layout for graph algorithms by grouping the graph elements logically close to each other close in memory. Using a similar optimization method, Burtscher and others [14] find that some applications significantly benefit from the optimized layout, while some other applications are not sensitive to the layout change. Comparing with these studies, we try to be general. Furthermore, they only consider data reordering, but overlook the benefit from data duplication.

Hardware Approaches: Rhu and others [77] design a locality-aware memory hierarchy that can automatically adjust the memory fetching granularity. For workloads of good spatial locality, the system fetches large chunks of data. For workloads of poor spatial locality, the system reduces the memory fetch width to save bandwidth. To reduce threads stall time, Meng and others [63] propose a threads subdivision scheme, which allows part of the SIMD group to continue the execution if the data for the involved threads are ready. It significantly reduces the memory latency imposed by irregular memory accesses.

The hardware thread scheduler also received some attention to improve memory performance. Rogers and others [78] design a cache-conscious warp scheduler. It explores the fact that exploring intra-warp locality is critical for irregular GPU applications. With the help from a intra-warp locality detector, the scheduler gives more exclusive accesses to the L1 data cache to the warps losing intra-warp locality. In their following work [79], proactive predictions help the scheduler schedule warps whose working set can fit into the L1 data cache.

The hardware approaches show great promise, but to our best knowledge have not been implemented in any off-the-shelf many-core processors.

1.3 Our Non-Uniformity-Aware Approach

We design a software framework centered around non-uniformity matching, which addresses three main limitations of existing approaches: 1) the quality-space overhead dilemma, 2) the quality-time overhead dilemma and 3) non-uniformity-oblivious thread scheduling. The first two limitations are relevant to run-time data reorganization, while the last one is a serious limitation from the GPU scheduler design. In the remainder of this section, we briefly describe those limitations and how this work addresses them.

1.3.1 Addressing the Quality-Space Overhead Dilemma

To reduce the number of off-chip memory transactions, modern GPUs have a key hardware optimization feature: *memory coalescing*. If the memory requests from one GPU SIMD group access the same memory segment (usually 128 byets), one memory transaction loads the whole memory segment to meet all the requests. However, the irregular memory access pattern caused by the mismatch of non-uniformity greatly degrades the benefit from memory coalescing. The full duplication algorithm proposed by Zhang and others [99], while guaranteeing to coalesce all memory accesses, introduces non-trivial space overhead. The space overhead throttles the benefit from data reorganization, as it increases the amount of data to be transferred from CPU to GPU. However, naively reducing the space overhead through partial duplication affects the layout quality, thereby degrading the final performance.

Our Contributions. We point out that the essence for designing an appropriate data reorganization algorithm can be reduced to a classical tradeoff among space, time, and complexity. Based on the insights, we develop two new data reorganization algorithms that complement prior algorithms with respective strengths. We

show that the new algorithms reduce space cost significantly with non-coalesced memory accesses kept minimized. We also compare the various algorithms, unify them into an assembly, and develop some selection guidelines and an automatic algorithm selector to address GPU dynamic irregular accesses in various scenarios. We experiment with a set of dynamic irregular applications and show that the developed assembly, along with the algorithm selector, circumvents the inherent complexity in finding optimal data layouts, making it feasible to minimize non-coalesced memory accesses for a variety of irregular applications and settings that are beyond the reach of existing techniques.

1.3.2 Addressing the Quality-Time Overhead Dilemma

The existing data reorganization algorithms have different levels of complexity. Typically, the more complex the algorithm is, the better quality the produced data layout has. However, complex algorithms incur large run-time overhead, which lies on the critical path and hence seriously limits their benefit. One tempting design is to offload data reorganization from critical path, which is, however, not feasible because of the data dependence between data reorganization and the irregular computation.

Our Contributions. We observe that the reorganization algorithm can be decomposed into two parts: transformation analysis and data repositioning. The former component figures out how to reorganize the data elements to match the non-uniform data interactions, while the latter component moves data to form the final data layout. The overhead of the analysis part is usually multiple times larger than that of the repositioning part. Fortunately, we can ensure the execution correctness as long as the data dependence between irregular computation and data repositioning is respected. Based on this insight, we offload the expensive analysis component to an originally idle processor and keep the lightweight data repo-

sitioning component on critical path. As such, we successfully hide data transformation overhead as well as guaranteeing the program's correctness.

1.3.3 Addressing the Non-Uniformity-Oblivious Scheduling

On current GPUs, the workloads processed by different threads show strong non-uniformity: 1) The data sharing degree varies across thread blocks and 2) Thread blocks from different kernels have different levels of memory access intensity. Nevertheless, the hardware scheduler, which is out of the reach of programmers or compilers, is unaware of such non-uniformity. As a result, we miss optimization opportunities from exploiting the non-uniformity of the hardware and the workloads. For example, scheduling threads that share much data to the same core can enhance inter-thread data reuses in the private data cache.

Our Contributions. We propose a compiler transformation and a runtime to enable flexible scheduling on modern GPUs. The key insight is that what matters is the mapping between jobs (the work a thread block processes) and the underlying hardware. By retrieving the IDs of cores during run-time, our framework flexibly determines which set of jobs should be processed by which processor. The flexibility enables several optimizations, including parallelism control, resource partitioning and affinity-based scheduling.

Chapter 2

Exploiting Non-Uniform Effects of Irregular References on GPU

2.1 Introduction

Recent years have seen a rapid adoption of GPU for high performance computing. As a massively parallel architecture, GPU significantly accelerates many regular, data-parallel applications. But its benefits for irregular applications are far less substantial, especially when the application contains dynamic, irregular memory references.

The reason comes from the hardware properties of GPU. GPU organizes its threads in groups and memory in segments. Every W threads with consecutive ID numbers form a *warp*; every S consecutive bytes in the global memory form a *segment*. At a memory reference, the number of memory transactions needed to load the data accessed by a warp equals the number of segments the data fall onto. When that number is larger than the possible minimum, the accesses are called *non-coalesced memory accesses*.

Non-coalesced memory accesses are common in irregular applications. Figure 1 (a) shows a simplified codelet in the core computation in a molecular dynamics simulation. The underlined statement "pos [neighbors [j*M + tid]]" gets

```
(a) codelet
      // tid: the global ID of a thread
      // M: num. of neighbors per molecule
      ipos = pos [tid];
      for (j=0; j< m; j++){
          jpos = pos [ neighbors [ j*M + tid]];
          computeForce (f, ipos, jpos);
      }
      force [tid] = f;
(b) case 1: neighbors [0...3] = {4, 5, 6, 7}
(c) case 2: neighbors [0...3] = {9, 103, 23, 67}
```

Figure 1: (a) A simplified codelet of the force computation in a molecular dynamics simulation. The values in *neighbors* decides the access pattern of *pos*. (b) and (c) show a regular and irregular pattern respectively.

the coordinates of a neighbor of the current molecule. As a typical dynamic irregular reference, it may manifest various access patterns, determined by the values contained in *neighbors*. In the case of Figure 1 (b), all accesses by the warp are to a single memory segment; only one memory transaction is necessary, assuming a segment can contain four molecules' positions. But in the case of Figure 1 (c), because of irregular values in *neighbors*, the accesses are non-coalesced and require four memory transactions. This kind of irregularity is common in a molecular dynamics simulation, thanks to molecules' movements and their dynamic neighborhood. It is a key feature of many scientific simulations.

Non-coalesced accesses may result in memory transactions as many as W times of the minimum, leading to a throughput a number of factors lower than the peak of GPU [9, 18, 99]. They have been the focused target of some recent studies. However, most prior explorations [9, 41, 56, 80, 97] concentrate on static irregularities, where the memory access patterns are known at compilation time. The type of irregularity in our focus is dynamic: For instance, the content of the indexing array *neighbors* in Figure 1 depends on the input to the program and is updated throughout the simulation of the molecules movement.

Dynamic irregular accesses have to be treated during runtime. Some earlier studies [87] have relied on special hardware extensions that modern GPUs do not have. A recent study [99] shows the promise of pure software solutions. It develops a pipeline scheme that makes it possible for CPU to reorganize data

and threads for a near-future GPU kernel invocation while GPU is working on the current kernel. A related study [18] demonstrates the feasibility of moving the reorganization to GPU so that CPU can also involve in workload processing.

Despite that these studies have shown promising speedups, the understanding to data reorganization for minimizing non-coalesced GPU memory accesses remains preliminary. Although it has been shown that minimizing the number of non-coalesced memory accesses is an NP-complete problem [93, 99], there only exist ad hoc algorithms to circumvent the complexity result. This chapter describes the first principled understanding of GPU data reorganization for minimizing non-coalesced accesses.

2.2 Problem Setting and Complexity Analysis

In this section, we first provide some background on GPU that closely relates with the following discussions. We then describe the main approaches researchers have been pursuing to tackle non-coalesced GPU accesses. We finally reveal the fundamental challenges for such approaches by proving that using those approaches to minimize non-coalesced accesses for general irregular references is *computational infeasible* unless NP equals P.

2.2.1 Background

As a massively parallel device, GPU contains hundreds or thousands of cores residing on a number of streaming multiprocessors (SM). When a GPU kernel is launched, the runtime usually creates thousands of GPU threads running on these cores in parallel. These threads are organized hierarchically. A number of threads (32 in NVIDIA GPU) with consecutive IDs compose *a warp*, a number of warps compose *a thread block*, and all thread blocks compose *a grid*. (This paper

uses CUDA terminology.) A warp is the unit in GPU scheduling; all threads in a warp proceed in lockstep.

GPU is equipped with several types of memory. The largest is off-chip main memory called *global memory*. It consists of a large number of segments (of 32, 64, or 128 bytes depending on the access mode.) For the large size and long access latency of global memory, its access efficiency is critical. GPU hence offers *memory coalescing*, a hardware-enabled feature that uses one memory transaction to load/store all the data in a memory segment that are requested by a warp at a load/store instruction. As a result, the execution of a load/store instruction by a warp incurs K memory transactions, where K equals the number of memory segments the requested data fall onto. Suppose the data to load/store by a warp at a reference contain D bytes and a memory segment is S-byte long. The reference is a *non-coalesced reference* when $K > \lceil D/S \rceil$. The corresponding memory accesses are *non-coalesced accesses*. Another type of memory on GPU worth mentioning is *shared memory*, which is on-chip with access latency comparable to that of register files. A thread can access an element that is loaded or stored into shared memory by another thread if and only if the two threads belong to the same thread block. In the remainder of the thesis, memory refers to global memory by default.

2.2.2 Objective and Complexity

The objective of non-coalesced access minimization is to minimize the number of non-coalesced accesses of a GPU kernel. The minimization, for its importance for GPU performance, has drawn lots of attentions. However, satisfying solutions are still limited to some special scenarios. In this section, we examine the inherent complexity of the previous approach and prove that in general cases, using the approach is infeasible to reach the objective unless NP equals P. The results may

guide the direction of future research, and also lays the theoretical foundation for the rest of this work.

A GPU kernel may contain multiple references. We focus on one reference first and discuss other scenarios later.

Data Repositioning and NP-Completeness

Data repositioning has been the main direction pursued by previous work for minimizing non-coalesced accesses [9, 56, 87]. The essential idea is to reorder data elements on memory so that the data to be accessed by a warp can reside consecutively, covering the minimum number of memory segments. For the example in Figure 1 (c), the transformation would create a new array *Pos'* with the same elements as *Pos* has but in a different order, such that the four elements *Pos[9], Pos[103], Pos[23], Pos[67]* fall into a single memory segment. Matrix transposing [87] is another example: By repositioning elements on memory to create a column-major data layout, it can minimize non-coalesced accesses for a column-wise reference to the matrix.

Although it seems simple, using data repositioning can be complicated when the data accessed by multiple warps overlap. Consider a reference $A[P[tid]]$, with P as follows

P[]={**8**, 23, 46, 93, **8**, **9**, 10, **67**, 5, 11, **41**, **67**, **9**, **41**, 55, 59}.

Assume memory segment length $S = 4$ and warp size $W = 4$. The repetitive values in P (highlighted in bold font) dictate that some elements in A are accessed by multiple warps. Which segment to put those values is tricky. For instance, putting A[8] into a segment with A[23], A[46], A[93] would coalesce the first warp's accesses but leave the second warp's accesses non-coalesced.

The issue has been largely limiting the applicability of data repositioning. Despite many recent efforts, this approach has been effective for only the cases where each target data element is accessed by only one warp in a kernel. In

an application with dynamic irregular references, that condition rarely holds: In a molecular dynamics, a molecule is often the neighbor of more than one molecules; in a sparse matrix multiplication, an element in the vector is often used to multiply multiple elements in the matrix; in a mesh simulation, a vertex is often shared by several triangles.

Complexity Theorem Prompted by the various difficulties people have so far encountered in finding a general data repositioning algorithm to guarantee minimum non-coalesced accesses, we conduct a systematic analysis on the inherent complexity of the problem. An important finding we obtain is that such an algorithm does not exist unless NP=P. Formally, we develop the following theorem:

Theorem 2.2.1 *Creating a new data layout through only data repositioning (which implies that each item in the original data structure has only one copy in the new structure) to minimize the non-coalesced accesses for an arbitrary data reference on GPU is an NP-complete problem.*

As this is the first strong claim on the complexity of non-coalesced access minimization, it is worth providing a formal proof, for verifying its correctness as well as offering insights that may be useful for analyzing the complexity of other GPU optimization problems.

Proof The proof is a result of a joint work, which was published in the dissertation of Zhang [98]. We elide the proof here.

When Warp Reorganization is Allowed

The above theorem assumes that only data repositioning is applied for reducing non-coalesced memory accesses. Some recent study [99] has shown that warp reorganization can help remove non-coalesced accesses as well, and can be used together with data repositioning for the optimization. In this subsection,

we complement Theorem 2.2.1 with Theorem 2.2.2, stating that reorganization does not change the NP-completeness of the problem.

Theorem 2.2.2 *It is an NP-complete problem to minimize the non-coalesced accesses for an arbitrary data reference on GPU through data repositioning, warp reorganization, or both.*

See the proof in [98].

2.2.3 Discussion

This section has analyzed the computational complexity of using data repositioning for minimizing non-coalesced accesses. The proved NP-completeness should not rule out the possibility that through some heuristic algorithms, the approach may still yield good speedup on some special types of kernels. However, it does indicate the extreme challenge to use it for achieving the optimal for general cases. We next show that the challenge can be circumvented if a constraint assumed by the approach is relaxed.

2.3 Algorithms that Circumvent the Complexity

We design two new algorithms to circumvent the complexity facing data repositioning. The key observation is that the essential difficulty in data repositioning comes from an implicit constraint that the produced new data layout uses no more space than the original. If we allow more space to be used, the complexity of the problem may reduce significantly.

Previous studies have not exploited this insight, except for the one by Zhang and others [99], which takes advantage of extra space but in an ad-hoc manner. In this section, we first review that previous method, reveal its limitation, and crystalize the analysis into an insight in the key tradeoff in designing a practical solution. We then describe the two new algorithms we design.

The following discussion is based on reference $A[P[tid]]$, a conceptual form of dynamic irregular references. It assumes the memory segment size (S) is a multiple of the working set size of a warp. This condition often holds given that the warp size and S are typically powers of 2. But even when it does not hold, the following discussions are still valid except that some preprocessing needs to be done to align data with memory segments.

2.3.1 Review of the *Duplication* Algorithm

The *duplication algorithm* is used by Zhang and others to optimize irregular memory accesses [99]. For an irregular reference, such as $A[P[tid]]$, the algorithm creates a new array A' such that $A'[tid] = A[P[tid]]$; the reference to $A[P[tid]]$ in the kernel is then replaced with $A'[tid]$. The algorithm naturally ensures that all accesses of a warp are to a consecutive memory region and there are no non-coalesced memory accesses, as illustrated by Figure 4 (b).

The algorithm is called "duplication" as it creates duplicated copies of a data element when the indexing array P contains repetitive values. Apparently, the new array A' is as large as the number of GPU threads (T), no matter how small the original array A is. Even worse is when there are multiple references to the same array (e.g., $A[P[tid]] + A[P[tid] + v]$), the algorithm creates a new T-long array for each of the references.

2.3.2 Limitations and Tradeoff

The duplication algorithm converts irregular accesses to regular ones. However, it may dramatically inflate space usage. For a K-element array referenced n times by T GPU threads, the space overhead is as much as a factor of $n * T/K$. In a modern GPU, T can be comparable with the number of bytes in the entire memory.

The large space overhead has two consequences. First, the basic duplication algorithm fails to apply when the space inflation exceeds the capacity of the

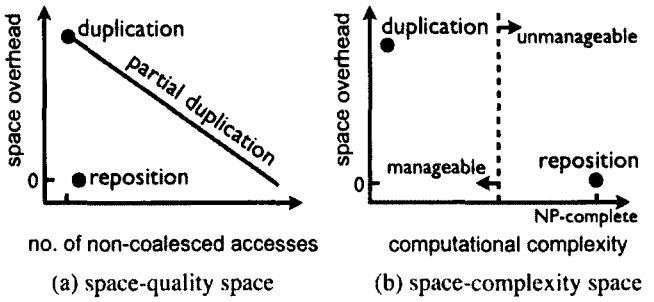


Figure 2: Positions of various algorithms in the space-quality-complexity coordinates. Graph (b) omits partial duplication for legibility.

memory. Second, the creation and transfer of the large volume of data may introduce substantial time overhead, throttling the optimization benefits. The previous work has used partial duplication to alleviate the issues [99]. The idea is to apply the transformation to only a fraction of the GPU threads. Although it can reduce the space overhead, it compromises the quality of the optimization proportionally. As Section 4.6 will show, it may result in substantially lower speedup than what is possible.

Figure 2 shows the conceptual positions of the previous approaches in a space of optimization quality, complexity, and space cost. Data repositioning and the duplication algorithm are at two extreme ends of the spectrum of space cost. The partial duplication lowers the space cost but also proportionally degrades the transformation quality and lengthens the kernel execution time. Data repositioning has the lowest space cost but the highest complexity. So the key for having a practical algorithm is to find a sweet design point that reduces the space cost without compromising the transformation quality and meanwhile possesses manageable complexity. We next describe two new algorithms towards that goal.

2.3.3 *Padding* Algorithm

The padding algorithm tries to avoid some unnecessary data copies made in the duplication algorithm without compromising the optimization quality. Its basic observation is that if two threads (t_1 and t_2) from the same warp access the same

data element (a), there is no need to create two copies of that data element. We can simply let them access the same copy of the data element. It will change the one to one regular mapping between data and threads created by the duplication algorithm, however, it will not create non-coalesced accesses since the two threads still access the same memory segment.

A Simple Design

A simple design is to just make the following modification to the duplication algorithm. When the algorithm is about to create a copy of an element in the new array A', it checks whether the current thread is the first of the current warp that accesses that element and avoids the creation if it is not. (An entailed change is that the original reference, say $A[P[tid]]$, needs to be replaced with $A'[Q[tid]]$ rather than $A'[tid]$, where Q contains the new mapping between a thread and the data it accesses in A'.)

Unfortunately, this simple modification is insufficient for two reasons. First, the avoided duplications cover only a small portion of all the duplications because the chance for two threads accessing the same data element to come from the same warp is often small. Second, the avoidance of some duplications often causes a misalignment between the working set of a warp and memory segments. As a consequence, the working set of a warp may span over the boundary of a segment, causing new non-coalesced accesses.

An Enhanced Design

Our enhanced design addresses the two problems of the simple design through sorting and padding. Figure 3 shows the pseudo code of the algorithm. It includes three steps.

The first step reorders data elements based on their access frequencies. At a data reference, the *access frequency* of a data element is the number of threads

```
// inputIndArray, outputIndArray: the original and produced indexing arrays
// inputArray, outputArray: the original data array and its new copy after padding
function Padding(inputIndArray, outputIndArray, inputArray, outputArray)
 inputIndArray = SortByFrequency(inputIndArray);
 for each warp w,
  uniqueRefSet = FindUniqueRefs(w, inputIndArray, inputArray);
  nRemainingSlots = FindRemainingSlots(currentMemSegment);
  if uniqueRefSet.size <= nRemainingSlots,
   for each e in uniqueRefSet,
    add element e to outputArray;
    update outputIndArray;
   end
  else
   pad dummy values to outputArray for memory segment alignment;
   for each e in uniqueRefSet,
    add element e to outputArray;
    update outputIndArray;
   end
  end
 end
end
```

Figure 3: The pseudo-code of the padding algorithm.

that access it. The second step reorganizes threads into warps. It reorders threads according to the order of data elements—that is, all threads accessing data X must precede all threads accessing Y if X precedes Y in the new data sequence. Starting from the first thread, every W adjacent threads form a warp in the resulting thread organization. These two steps address the first problem of the simple design: By making threads accessing the same data element locate closely and form warps, they reveal more opportunities for saving duplications.

The third step puts data elements into memory segments. Starting from the first data element in the new order, the data are greedily packed into a memory segment one after one. But when it finds that the current segment cannot hold all the data the current thread warp accesses (e.g., the first segment in Figure 4 (c)), it moves all the data accessed by that warp into the next memory segment, leaving some empty slots at the end of the previous memory segment. Data is duplicated only when necessary—that is, when one data element is accessed by multiple thread warps whose working data sets do not fall into the same memory segment. Examples are the two "c"s in the layout in Figure 4 (c). This step addresses the second problem of the simple design. By padding a memory segment with some

threads: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
accessed object: a a b b c c d d e e a a b b c d e f g h a a ...
original layout: a c e g b d f h ...
acc freq: 6 3 3 1 4 3 1 1

(a) Original layout and accesses

new layout: a a b b c c d d e e a a b b c d e f g h a a ...
acc freq: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

(b) Layout from Duplication

new layout: a b c x c d e f g h ...
acc freq: 6 4 2 0 1 3 3 1 1 1

(c) Layout from Padding ("x" for empty slot)

new layout: a b c d e f g h ...
acc freq: 6 4 3 3 3 2 1 1

(d) Layout from Sharing

Figure 4: An example that illustrates the algorithms of duplication, padding, and sharing. Assume 4 objects per memory segment, 4 threads per warp, and 4 warps per block.

empty slots when necessary, it aligns the working set of a warp with memory segments.

Analysis This padding algorithm guarantees zero non-coalesced access since it puts the working set of every warp into a single memory segment. Its space overhead comes from the padded empty slots and some duplicated data elements. If k threads in a warp access one single data element, the empty slot in a memory segment is at most as long as $mod(S, \lceil W/k \rceil)$, where $\lceil W/k \rceil$ computes the number of unique elements accessed by a warp and S is the number of data elements a segment can contain. Both S and W are usually power of two. So the worst case happens when k is small (hence the remainder is large) but is not a power of two. Specifically, when $k = 3$, the empty slot is the longest, up to $W/3 - 1$. But even in that case, the space cost is much lower than that of the duplication

algorithm. The number of threads a memory segment serves in that case must be no fewer than $t = (S - W/3 + 1)/(W/3)$. Following the assumption that S is the multiple of W, let $S = r * W$ with r being a positive integer. The number of threads served, t, must be no smaller than $3r - 1$, which is at least 2. In the duplication algorithm, these threads would use at least $2S$ memory (given that $r = 1$ means $S = W$), double what they use in the padding algorithm.

When analyzing the number of duplications in the padding algorithm, it is important to notice that among all the warps accessing the current memory segment, only the first of them may have some data elements duplicated. It is because only when the working set of a warp overlaps with the data elements in the previous memory segment, those overlapped elements may have a duplicate in the current memory segment (e.g., the second "c" in Figure 4 (c).) That overlap must be partial since at a complete overlap, the previous memory segment can already hold the working set of that first warp, and hence that warp would have corresponded to the previous rather than current memory segment. Due to the way threads are ordered, that partial overlap entails that the working sets of the other warps cannot overlap with the data in the previous segment, and hence have no duplicated data. Following the observation, we can see that in the case mentioned in the previous paragraph, the duplicated part of a segment is at most $W/3 - 1$, smaller than $1/3r$ of S. In comparison, the duplication algorithm creates at least 3 copies per data element in that case. The data element contained in one memory segment in the padding result would become $3 * (S - W/3 + 1)/S$ (which is greater than $3 - 1/3r$ and 2.67) segments in the result from the duplication algorithm.

Limitation Despite its appealing properties, the padding algorithm has one major limitation. Because it reorganizes not only data but also threads, it may cause side effects to other references in the kernel. For example, if a kernel contains "B[tid]+A[P[tid]]", after the third and ninth threads switch positions, they swap their

jobs, and the accesses to B must also be swapped. In another word, $B[tid]$ must be replaced with $B[R[tid]]$ where, $R[3] = 9$, and $R[9] = 3$. Otherwise, the new thread 3 would add $A[P[9]]$ with $B[3]$ rather than $B[9]$, causing wrong computation results. As a result, the optimization of $A[P[tid]]$ makes accesses to B non-coalesced. So the padding algorithm is most beneficial when all references in a kernel follow the same access pattern (e.g., $B[P[tid]] + A[P[tid]]$.)

2.3.4 *Sharing* Algorithm

This algorithm overcomes the limitation of the padding algorithm by increasing duplication avoidance from a different angle. It uses the shared memory in GPU to enlarge the scope of duplication avoidance. Shared memory is a type of on-chip memory in GPU. Data written to shared memory by a thread is visible only to threads in the same thread block. Shared memory has an access latency a hundred times smaller than that of the global memory, and more importantly, its performance is largely insensitive to irregularities in accesses.

Insight The key insight of this algorithm is to shift irregular accesses from global memory to shared memory. As shared memory is visible within a whole thread block, the sharing algorithm enlarges the scope of duplication avoidance from a warp to a thread block. Its basic idea is to create a copy of all the data accessed by a thread block (a single copy per data element) and put them into a consecutive chunk of memory. Then, it loads these data in a consecutive (hence coalesced) manner into shared memory. It redirects memory accesses by the thread block to the corresponding copies in the shared memory. By keeping only one copy for all data elements accessed by a whole thread block, it avoids many duplications. By shifting irregular accesses to shared memory, it eliminates non-coalesced accesses to global memory. It uses clustering to further increase the opportunity for duplication saving. The detailed algorithm is as follows.

```
// inputIndArray, outputIndArray: the original and produced indexing arrays;
// inputArray, outputArray: the original and produced data arrays;
// blockPos, blockSizes: the starting position and size of the working set of a thread block
function Sharing(inputIndArray, outputIndArray, inputArray, outputArray,
blockPos, blockSizes)
  [inputIndArray, inputArray] = DataClustering(inputIndArray, inputArray);
  for each thread block b,
    uniqueRefSet = findUniqueReferences(b, inputIndArray, inputArray);
    blockSizes[b] = uniqueRefSet.size;
    blockPos[b] = outputArray.size;
    for each e in uniqueRefSet,
      Add element e to outputArray;
      outputIndArray[e] = position(outputArray, e) - blockPos[b];
    end
    pad dummy values to outputArray for memory segment alignment;
  end
end
```

Figure 5: The pseudo-code of the sharing algorithm.

Algorithm Figure 5 outlines the pseudo-code of the sharing algorithm. It includes two steps. In the first step, it conducts clustering to swap threads among thread blocks so that the threads in a block have as many accesses to the same data elements as possible, while different blocks have as few as possible.

Many clustering algorithms can serve for the purpose. In our implementation, we use two. The first is a graph partition-based clustering [92], which is especially suitable for applications with a graph topology, such as the distribution of molecules in a molecular dynamics simulation, the structure of a mesh in a mesh simulation. In these applications, typically each thread is in charge of one node in the graph. The algorithm randomly selects some nodes as seeds and assign each of them a distinct cluster number. The nodes then iteratively propagate the cluster memberships to their neighbors. The threads are clustered by inheriting the cluster ID of their corresponding nodes. The second clustering algorithm is suitable for other cases. It uses the working set of a thread as its feature and applies the standard hierarchical clustering to build up the clusters.

After clustering, the second step prepares data to be loaded into shared memory and creates a new indexing array to reference them. Specifically, it places the data elements accessed by each thread block continuously into a global array. It is possible that even after the clustering, the working sets of two thread blocks may still overlap. In that case, some data will have to be duplicated across thread

blocks. Some trivial padding fills up the final memory segment a thread block uses. Figure 4 (d) shows an example. The clustering switches threads 9 and 10 with threads 21 and 22. After that, each thread block accesses four unique data elements and there is no overlaps between the working sets of the two blocks and hence no duplications. Two meta-arrays, *blockPos* and *blockSizes*, record the starting offset and the number of accessed data elements in the new data array for each thread block. They add minor space cost. When the GPU kernel executes, each thread block loads the corresponding block of data to shared memory according to the meta-arrays.

Notes We make several notes. First, the clustering step is optional. It increases the chance for saving data duplications, but even if it is not used, the algorithm can still remove all non-coalesced accesses and avoid duplications inside a thread block.

Second, when clustering is used, threads in different blocks may get swapped. However, unlike the padding algorithm case, even with the swapping, the sharing algorithm can still apply to a kernel containing multiple references with different access patterns. It is because the sharing algorithm does not require data references to remain or become regular. Consider the example mentioned earlier, $B[tid] + A[P[tid]]$. After clustering-incurred thread swapping, the references may become $B'[Q[tid]] + A'[P'[tid]]$ and both references become irregular. However, the second step of the sharing algorithm ensures that both arrays will be loaded into the shared memory in a coalesced manner. Accesses to the copies in the shared memory may be irregular, but recall that the performance of shared memory is resilient to access irregularity. It is worth noting that for this algorithm to work properly, the clustering and data reorganization need to put all the references ($B[tid]$ and $A[P[tid]]$ in our example) into consideration.

Third, the usage of shared memory may have two side effects on the performance of the kernel. The first is the time overhead of the introduced accesses to shared memory, which is often negligible compared to the time incurred by global memory accesses, especially for the irregular applications that are often memory latency bound. The second effect is that because shared memory is partitioned to all active thread blocks on a streaming multiprocessor, a large usage of shared memory by a thread block may reduce the number of thread blocks that can be active at the same time (called GPU *occupancy*.) Our experimental results in Section 4.6 show that the effect is not obvious on irregular applications.

Finally, when a problem size is large, the working set of a thread block could be larger than the shared memory. Fortunately, we observe that for most kernels, when the problem size increases, the problem size per thread block often remains unchanged but more thread blocks are created. In exceptional cases, to apply the sharing algorithm, the kernel can be modified to break the task of one block into smaller tasks and assign them to more thread blocks.

Analysis Quality-wise, as described in the algorithm, after the sharing algorithm applies, the accesses to the global memory become consecutive and coalesced. It maintains the zero non-coalesced accesses guaranteed by the duplication algorithm.

Space-wise, the algorithm saves space cost by avoiding data duplications for threads inside a block. The maximum number of copies of a data element is the number of thread blocks, rather than the number of threads in the duplication algorithm. If on average k ($k < B$, B for the number of threads per block) threads access one data element, with a perfect clustering that puts threads accessing the same data element into a single block, the algorithm can virtually avoid all data duplications. In practice, the amount of savings depends on the clustering

quality (or how frequently multiple thread blocks access the same data elements if clustering is not used.) Section 4.6 provides the empirical results.

2.3.5 Discussion

The two new algorithms introduced in this section guarantee zero non-coalesced access as the duplication algorithm does. Although they reduce the space overhead of the duplication method substantially, it should be noted that they do not guarantee minimum space cost. Designing an algorithm with that guarantee and zero non-coalesced accesses is not the goal of this work. In fact, that task is no easier than the data positioning problem (they form dual problems with each other.) Section 4.6 will show that the two algorithms do provide practical solutions to a variety of programs.

2.4 Evaluation

In this section, we evaluate the proposed algorithm assembly on eight benchmarks in Table 1, which all have dynamic irregular memory accesses. For comparing with the state of the art [99], they include all memory benchmarks used in the previous work: CFD is an unstructured grid finite volume solver; CG is a conjugate gradient method with sparse matrix-vector multiplication as its kernel; NN is for nearest-neighbor clustering; UNWRAP is for 3-D reconstruction. MD is a molecular dynamics simulation from the Shoc benchmark suite [22]; NBF and IRREG are derived from two irregular CPU applications heavily used by previous research [37]. The former is part of GROMOS, a force field simulation; the latter is the core of an iterative partial differential equation solver. The benchmark MERGE is a database update program. All code has gone through performance tuning to fit the execution models of GPU. The inputs to MD, IRREG, NBF and CFD consist of some nodes and neighbor lists generated randomly. The input to

Table 1: Benchmarks and selected optimization algorithms

(M1:Tesla C1060; M2: GTX480)

benchmark	description	alg. on M1	alg. on M2
MD	molecular dynamics	Sharing	Sharing
IRREG	partial diff. solver	Sharing	Sharing
NBF	force field	Sharing	Sharing
CFD	finite volume solver	Sharing	Sharing
CG	conjugate gradient	Sharing	Sharing
UNWRAP	3-D reconstruction	Dup.	(not runnable)
NN	nearest neighbor	Dup.	Dup.
MERGE	database update	Padding	Padding

unwrap cannot run on GTX480 for unknown reasons.

Table 2: Transformation ratios

benchmark	Dup.		Sharing	
	C1060	GTX480	C1060	GTX480
MD	0.25	0.1	0.85	0.65
IRREG	0.4	0.1	0.9	0.7
NBF	0.4	0.15	0.95	0.8
CFD	0.35	*	0.6	*
CG	0.45	0.15	0.5	0.2
UNWRAP	1	-	1	-
NN	0.7	0.4	0.7	0.4
MERGE	0.3	0.3	0.6	0.6

*: optimization is shut down; "-": not runnable.

MERGE includes some indexing arrays of a set of data generated randomly. The inputs to CG contain a sparse matrix and vector. The locations of the non-zero elements in the matrix exhibit some patterns such that multiple rows of the matrix happen to multiple with a similar set of elements in the vector. The inputs to NN and UNWRAP come with the benchmarks.

We experiment on two types of GPU devices. One is NVIDIA Tesla C1060 hosted in a quad-core Intel Xeon E5640 machine, and the other is NVIDIA GTX480 hosted in a quad-core Intel Xeon E5520 machine. Both machines have CUDA4.2 installed. We obtain hardware performance through the NVIDIA's CUDA profiler.

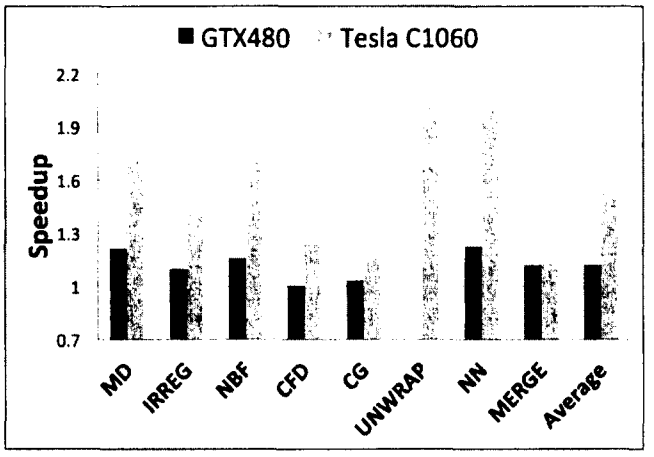


Figure 6: Speedup of selected algorithms

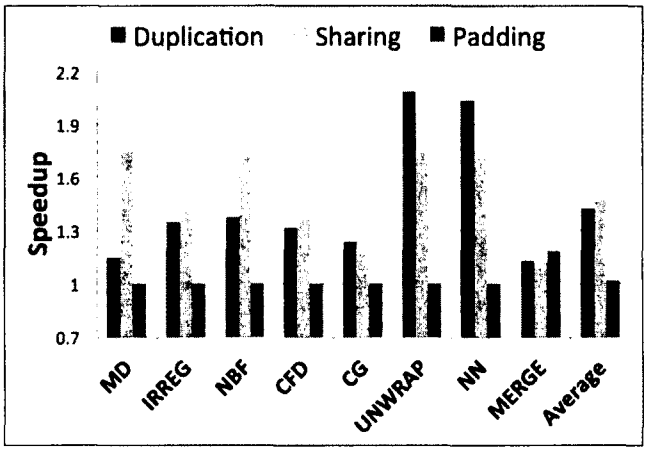


Figure 7: Speedup of all algorithms (Tesla C1060)

Results Overview Figure 6 reports the kernel speedups on both machines with the baseline as the execution time of the original GPU version. All overhead, including transformation and extra data transfer, is included. The selector-based algorithm assembly produces up to 21% speedup on GTX480. It gives even larger speedup (up to 109%) on C1060 because that device is more sensitive to irregular accesses for its lack of on-chip cache. (It is worth noting that having cache or not on massively parallel processors is still a debatable topic; some recent chips, such as Intel SCC, do not have cache for power efficiency.)

For further confirmation, we use the NVIDIA hardware performance profiler to measure the memory load efficiency. Load efficiency is defined as the ratio of requested global memory load throughput to actual global memory load through-

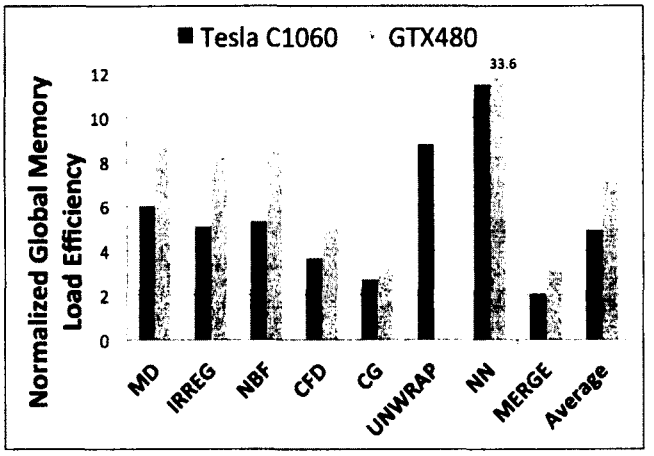


Figure 8: Memory load efficiency of selected algorithms.

put. As Figure 8 shows, the algorithm assembly improves the average efficiency by 4.9X on C1060 and 7.2X on GTX480 over the original version.

The two rightmost columns of Table 1 report the selected algorithms on the two machines. Figure 7 shows the speedups brought by each algorithm on Telsa C1060, confirming that all selections except for the one for CG on Telsa C1060 are correct. (We explain the selection error later in the detailed analysis on CG.)

Six of the eight benchmarks benefit the most from the newly designed algorithms on at least one machine. As Figure 7 shows, the new algorithms produce extra speedup as much as 8-60% over the duplication algorithm. It is mainly due to the larger transformation ratios (shown in Table 2) enabled by their large reduction of the overhead in data copy and transfer. The padding algorithm, due to its constraint on access patterns, is applicable to only the MERGE benchmark in the suite.

Overall, the results show that the two new algorithms significantly enhance the power of data reorganization for irregular memory optimizations. The algorithm assembly and online selector produce some promising speedups for most of the benchmarks. We next discuss each benchmark in further details.

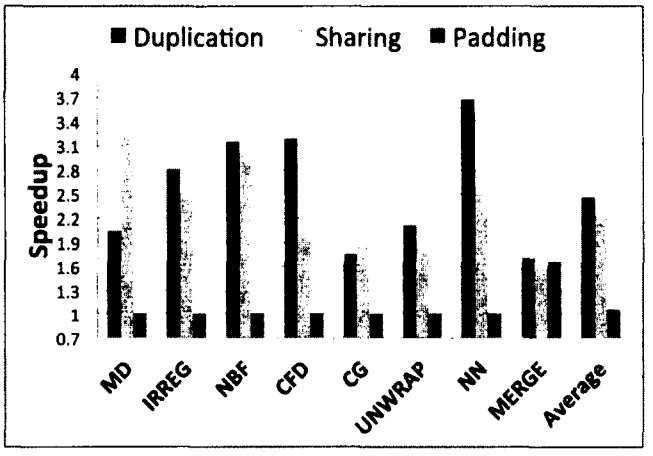


Figure 9: Potential speedups of all algorithms (Tesla C1060)

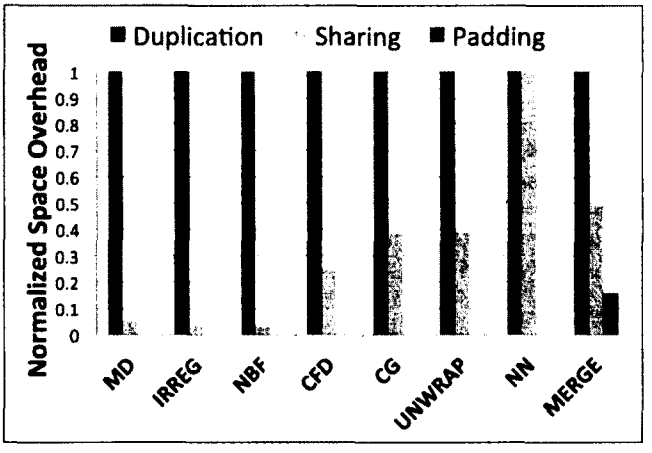


Figure 10: Normalized space overhead (*padding* is only applicable to MERGE.)

MD, IRREG and NBF MD simulates the interactions of a number of molecules in a 3-D space. Two molecule nodes are neighbors if their distance is smaller than some predefined threshold. One thread is in charge of each node. In a simulation iteration, that thread traverses all its neighbors to calculate the force between each neighbor and that node. The inefficient memory references come from reading neighbors' positions.

The duplication algorithm improves the performance by duplicating position values to make sure adjacent threads load adjacent memory locations. Figure 9 shows that the full duplication can give 2X speedup on C1060 when overhead is not counted. But the overhead of data creation and transfer throttles the speedup to only 15%. The sharing algorithm has a higher performance potential than the duplication algorithm for the smaller working sets. Figure 10 reports that the sharing algorithm cuts the space overhead by 96%, which explains the seven times more speedup it creates than the duplication algorithm does when overhead is counted as Figure 6 shows.

The tremendous space reduction comes from two reasons. First, the irregular reference to data array is surrounded by a loop to traverse all neighbors, and in each iteration the memory access pattern of all threads is different depending on the topology of the interaction graph. The duplication algorithm, therefore, duplicates the same array the same number of times as the iteration number. Second, Sharing benefits greatly from clustering, which places adjacent nodes in topology closely in memory accessed by threads in the same block.

IRREG and NBF, like MD, have a graph topology. Figure 9 shows different potentials, because their kernels have different ratios of computation to memory accesses. Nonetheless, the sharing algorithm is also shown to be the best choice for them due to the reasons similar to MD. It is worth mentioning that the benefits of the optimizations also depend on the frequency of the neighbor list update in these simulation programs. When the update is frequent, the data transformations need

to be applied often and hence lead to higher transformation overhead. When the overhead cannot be hidden completely, the runtime control of G-Streamline can adaptively adjust the fraction of data to transform to trade data layout quality for transformation efficiency [99]. A detailed study on various tradeoffs of the different algorithms in these frequent update scenarios are our future work.

CFD The program, CFD, computes force field of many particles. Each particle has substancially more features than the molecules in MD, and so each thread block processes more data. Figure 9 shows a potential of more than 3 times speedup from the duplication algorithm. But the data transfer overhead throttles the potential. The algorithm eventually produces 31% benefit with a 0.35 optimization ratio on C1060. The smaller space overhead of the sharing algorithm leads to a larger optimization ratio (0.6) and a higher speedup (37%).

CG The kernel in CG does sprase matrix multiplication. The matrix is stored in the Compresses Sparse Rows (CSR) format. In the irregular kernel, one thread is in charge of one non-zero element in the sparse matrix. The accesses to the vector may not be coalesced depending on the sparsity in each row. The duplication and sharing perform similarly well in the potential graph. The best speedup on C1060 is 1.85 times, while the performance gain is around 20% on GTX480 due to the cache effects on the reuses of the elements in the vector. Figure 9 reports that the sharing algorithm has slightly larger potential than the duplication on C1060. The better algorithm, however, is duplication, because of the overhead caused by shared memory accesses. The subtle difference is not captured by the online algorithm selector, causing the sharing algorithm being selected. But the speedup lost is only less than 5%.

UNWRAP The kernel of this program is in a central loop, which transforms an image from the Cartesian coordinate system to the Polar coordinate system. Un-

like the other programs, this program do not have data dependences carried by the different invocations of the kernel. The first tens of iterations of the program successfully overlap the overhead of both the duplication and sharing algorithms. The duplication was shown to be the better algorithm for its lack of the side effects in shared memory usage. (For unknown reasons, the benchmark cannot run on GTX480.)

NN The nearest neighbor identification program, NN, has a central loop to process an unstructured input file chunk by chunk. The kernel is launched once for each chunk, and calculates the Euclidean distances from the target location to each record in that chunk. At the end of the program, the main thread processes all distance results and obtains the K nearest neighbors. Figure 9 shows the large speedup potential on both C1060 and GTX480. The sharing algorithm does not reduce any space overhead as reported in Figure 10. The reason is that one record is only processed once, and the duplication algorithm essentially just transposes the data, creating no extra data copies. Like UNWRAP, there is no loop-carried dependence for NN, but the transformation and transfer overhead can not be fully overlapped, and we obtained 0.7 and 0.4 optimization ratios on C1060 and GTX480 respectively. On this special benchmark, the duplication algorithm is a better algorithm in both machines, producing higher speedup than sharing.

MERGE MERGE has the same access pattern for both loads and stores. Padding algorithm is applicable. As Figure 9 shows, Duplication and Padding have quite similar potential. Padding has a larger potential than Sharing because it needs no shared memory accesses. Padding reduces the size of transformed data significantly. Duplication, due to the memory size limit, only manages to transform 30% of data. Padding is the best choice on both GPUs for this program. The speedups on the two machines are quite similar on this program. The reason is

that the program has few short reuse distances and hence does not benefit from cache much.

2.5 Algorithm Selection and Integration

The three algorithms described in the previous section have different strengths and weaknesses. In this section, we provide a qualitative comparison, and describe an automatic selector and its integration in a runtime library.

Qualitative Comparison We summarize the qualitative differences among the three algorithms as follows.

- **Applicability:** The padding algorithm is applicable to kernels that have a single reference pattern. While the other two algorithms do not have such a constraint, the sharing algorithm may need kernel modification when the working set of a thread block is too large to fit into shared memory, and the duplication algorithm may be applicable to only part of the data when the space limit is reached.

- **Space cost:** By avoiding unnecessary duplications, the padding and sharing algorithms use much less space than the duplication algorithm does.

- **Optimization capability:** All three algorithms have the capability to eliminate all non-coalesced memory accesses (in their applicable scenarios.) However, when being applied at runtime, the realizable benefits also depend on their runtime overhead.

- **Transformation overhead:** The time overhead of the duplication algorithm is in the creation and transfer of the new data copies, which can be substantial when the number of threads is very large or there are multiple references of different patterns to the same array. For the padding algorithm, the overhead

```
// T: # of threads;  D: the set of memory references;
// D_1: working set of a thread block;
// Z: the size of the irregularly accessed data;
if T is less or comparable with Z:
   use duplication;
else if D has a single access pattern:
   use padding;
else if D_1 is smaller than shared memory:
   use sharing;
else:
   use duplication or change kernel to use sharing.
```

Figure 11: Guidelines for algorithm selection.

includes the data and threads sorting time in addition to the creation and transfer of the new arrays. The overhead of the sharing algorithm consists of data creation and transfer time, the clustering time, the accesses to shared memory and the side effect on occupancy. Due to the large space reduction, the data creation and transfer in the two new algorithms usually take much less time than in the duplication algorithm. Data creation and transfer usually reside on the critical path of dynamic simulation applications, but the sorting and clustering in those two algorithms do not and hence can be largely hided (e.g., through the CPU-GPU pipeline in G-Streamline [99].) We will come back to this point later in this section.

Algorithm Selection Based on the differences, we develop some simple guidelines, as Figure 11 shows, to help programmers select the suitable algorithm when writing a new program.

Meanwhile, we provide an automatic selector based on the online profiling and adaptive control offered by G-Streamline, a runtime library we previously developed [99]. The runtime library works when the GPU kernel is invoked in a loop. By profiling the initial several iterations during runtime along with some performance models of the system (e.g., the time to transfer a data from CPU to GPU, the time to create a data copy) built ahead of time through offline profiling,

it estimates the kernel running time and optimization overhead to determine the suitable optimization algorithm to apply and the appropriate optimization parameters to use (e.g., the fraction of data to optimize in partial duplication.) Many irregular applications, including dynamic simulations and numerical calculations, are of that iterative pattern and are amenable for the runtime library to work. Our automatic selector employs the online profiling to estimate the amount of overhead of the algorithms and the kernel running time to pick the algorithm with the largest performance potential.

Integration with G-Streamline We integrate the selector and the reorganization algorithms into G-Streamline. G-Streamline uses a CPU-GPU pipelining scheme to allow runtime optimization of a future kernel invocation to happen on CPU when GPU is running the current invocation. However, if the future kernel's optimization depends on its previous invocation result, the optimization has to happen on the critical path. In that case, to make the optimization still happen asynchronously, kernel splitting is used so that the computations of a kernel are split and put into two parallel sub-kernels. The optimization of the second sub-kernel can run with the invocation of the first sub-kernel. The ratio between the amount of task between the second and first sub-kernel is called *transformation ratio*. The more costly the optimization is, the lower the ratio has to be so that the invocation of the first sub-kernel can hide the optimization overhead.

For all irregular applications we find, among the major operations in the three algorithms, sorting and clustering can happen across kernel invocations, but data creation and transfer are on the critical path due to dependences across kernel calls. They have to rely on kernel splitting to hide their overhead. In Section 4.6, we will see that the padding and sharing algorithms have much higher transformation ratio than duplication for their much smaller overhead in data creation and transfer. It is worth noting that G-Streamline uses its online profiling scheme to

determine the suitable transformation ratio to ensure all overhead is hidden. If an optimization is infeasible to give benefits, G-Streamline shuts it down automatically to prevent any slowdown to the kernel.

Integrating the data reorganization algorithms into G-Streamline does not change the library's interface. It only adds a handful of functions as alternatives to the duplication algorithm already presenting in G-Streamline. The usage of the modified G-Streamline is the same as before [99]: Users insert several function calls into the GPU application to invoke the runtime asynchronous optimizations and online profiling; some minor changes to the kernel may be needed, including replacing old indexing arrays with new ones.

2.6 Related Work

Sections 2.1 and 4.6 have compared this work with previous studies [18, 87, 99] on optimizing dynamic irregular memory accesses on GPU. This section reviews some other related studies.

A number of studies have proposed compiler techniques to optimize GPU memory references. Examples include GPU optimizing compilers [41,97], OpenMP-to-CUDA compilers [56], polyhedral models [9], performance tuning [80], and many others that cannot be listed for lack of space. All these techniques have focused on static irregularities that are amenable for compiler analysis. The usage of shared memory in the design of our sharing algorithm is inspired by some of those previous work [9]. But to our best knowledge, the sharing algorithm is the first algorithm that uses clustering and shared memory to avoid data duplications for runtime data reorganizations.

There are some recent studies exploring the synergistic usage of CPU and GPU, including the execution strategies proposed by Huo and others [43], the exploitation of OpenCL [51], and so on. In this work, we use the CPU-GPU pipeline

created in G-Streamline [99] as it meets the needs for hiding transformation overhead.

Thread divergence is another type of dynamic irregularity in GPU, defined as the threads in a warp follow different paths of a kernel. Some hardware extensions have been proposed to remove thread divergences from a kernel execution [31, 63]. Carrillo and others [15] have proposed loop splitting and branch splitting to alleviate register pressure caused by diverging branches. As pointed out by an earlier work [99], thread divergence and non-coalesced memory accesses essentially stem from the similar source, a mismatch between threads and data. It suggests that the findings from this study are potentially usable for helping eliminate thread divergences as well.

Data reorganization has been used for many CPU data locality enhancements (e.g. [5, 16, 20, 28, 48, 96].) Some of them have especially concentrated on irregular applications [37, 85]. Kulkarni and others have studied locality issues of irregular data structures in the contexts of optimistic parallelism [53] and scheduling [45]. As a massively parallel architecture, GPU displays different memory access properties from CPU, exemplified by the hierarchical thread organizations, hardware enabled memory coalescing, and the SIMT execution model. All these features create differences in the challenges and opportunities in applying data reorganization, triggering the new set of innovations in this work on both complexity analysis and transformation techniques.

2.7 Summary

This chapter presents some fundamental understanding in exploiting data reorganization for minimizing non-coalesced memory accesses on GPU. It points out that it is possible to circumvent the complexity by relaxing the space constraint in data repositioning. It introduces two new algorithms for minimizing non-coalesced

memory accesses while avoiding the space inflation problem of a previous algorithm. It compares the various algorithms, presents some selection guidelines, and develops an automatic selector in a runtime library. Experiments show that the new algorithms excel previous techniques especially under space pressure. The algorithm assembly, assisted by the algorithm selector, enhances the performance of a set of dynamic irregular applications significantly, providing promising solutions to a large class of dynamic irregular references.

Chapter 3

Capitalizing Non-Uniform Data Affinity by Overcoming Data Dependences

3.1 Introduction

Due to the memory wall problem on traditional architecture, data locality has been one of the most prominent factors that determine the performance of a program. Its importance is even more pronounced on modern Chip Multiprocessors (CMP), where, the last-level cache and memory bus bandwidth are typically shared by multiple cores. The sharing causes contention among co-running applications, and the effect intensifies as the number of cores grows on a chip. Data locality enhancement is an important approach to tackling the problem. It reduces the required accesses to the last-level cache and memory to alleviate the pressure on shared memory hierarchy.

Special difficulties for locality enhancement come from irregular memory references. Such references often arise in dynamic simulation applications—such as unstructured mesh simulation and molecular dynamics simulation—due to the use of sparse and irregular data structures. A representative form of irregular references is $A[P[i]]$, where the index array P may be embodied in real applications

by an input array or intermediate computation results that are difficult to know until run time.

Irregular memory references have several properties. First, because the content of the index array P may be arbitrary, the references to A tend to cause serious locality issues. Second, the memory access patterns of those references are unknown until execution time. Third, applications having such references tend to contain a main loop—such as, the mesh refinement loop in mesh generation, the time elapse loop in molecular dynamics simulation—that encloses the irregular memory references. The access patterns of the references (e.g., the values in P) often vary across the loop iterations. These properties make locality enhancement of irregular references extremely difficult for static compilation techniques.

A number of prior studies [28, 37, 62, 65, 85] have pursued runtime data transformations to attack dynamic irregular references. The strategy is to reorder data objects *during an execution* based on their exhibited access patterns.

However, the power of the prior transformations has been restrained by a dilemma. In all prior techniques, the runtime data or computation reordering happens synchronously—that is, the reordering is on the critical path of the application. This feature results in a tension between transformation quality and runtime overhead: More sophisticated transformations often yield better locality and save more execution time, but at the same time, they add more transformation overhead to the overall execution. The overhead can be substantial, especially for sophisticated transformations. For instance, one application of RCB—a classic data transformation approach—takes more than 20 simulation time steps in most experiments reported in Section 3.7. Moreover, the transformation have to be applied repetitively due to the iterative computations in dynamic simulations. Some studies propose to apply the transformation occasionally rather than everytime when access pattern changes [36, 61]. Unfortunately, it is subject to the same

quality-overhead dilemma: The less frequently the transformation applies, the less overhead it causes, but the worse the data layout is.

In this chapter, we propose three orthogonal techniques to resolve the quality-overhead dilemma.

The first is *asynchronous data transformation*, supported by a dependence-circumventing decomposition. The basic idea is to hide the transformation overhead by offloading the main transformations from the critical path, making them happen asynchronously (on an idle processor) in parallel with the execution of the application. Despite the simplicity of the idea, to the best of our knowledge, asynchronous data transformation has not been proposed previously. The plausible reason exists in the circular data dependences between data transformations and the execution of the application. On one hand, the transformation modifies the data structure that the application needs to read; on the other hand, the transformation needs to read some results computed by the application to figure out the appropriate data order. So, inherently, one invocation of a data transformation must run serially with the corresponding iteration of the application. In this work, we circumvent the problem by decomposing data transformation into two parts and safely relaxing some dependences through a careful analysis and layout approximation.

The second technique we develop aims at overhead minimization, especially for a system equipped with massive parallel devices (e.g., GPU). We propose a novel data transformation algorithm, named *TLayout* (*T* for throughput), which reduces transformation overhead significantly with little compromise to the resulting quality. Unlike traditional data transformation algorithms, TLayout is a massively data-parallel algorithm, specially customized to the strengths of throughput-oriented co-processors. It is novel in using an almost dependence-free approach to grouping nodes into a number of clusters such that the nodes referenced adja-

cently fall into the same cluster. The algorithm shows high efficiency and scalability.

Asynchronous data transformation and TLayout tackle the limitations of previous data transformations in two orthogonal directions; one for overhead hiding, the other for overhead minimization. Together, they help resolve the quality-overhead dilemma that prior approaches have been facing.

The third technique we develop is an online adaptive scheme. By transparently selecting the appropriate transformation strategy during runtime, the scheme gains the best of both asynchronous and synchronous transformations, proving able to overcome the limitations of both strategies.

Overall, the proposed techniques yield 65% higher performance improvement than previous techniques do, accelerating the original dynamic simulations by as much as a factor of 3.1 (2.4X on average) on five representative dynamic simulation benchmarks.

3.2 Background on Irregular References and Runtime Locality Enhancement

Irregular references commonly exist in dynamic simulation programs due to the use of sparse and irregular data structures. They are typically in forms of indirect references like $A[P[i]]$.

Previous solutions to irregular references use runtime data and computation reordering. In computation reordering, the iterations of the central computation loop are reordered so that the iterations accessing the same or adjacent data elements are adjacent in time. This transformation requires that there are no dependences across the iterations. Techniques for determining the suitable iteration order include lexicographical sort [24], bucket sort [65], z-sort [37], and so on.

Data reordering repositions elements in an array to improve spatial locality. The basic strategy is to relocate the elements such that the elements that tend to be accessed closely in time become close in memory space. Because determining optimal data orders is an NP-hard problem in general [73], researchers have proposed various heuristics-based algorithms, including consecutive packing (CPACK) [28], Reverse Cuthill-McKee (RCM) [58], space filling curve (SFC) [62], recursive coordinate bisection (RCB) [12], multilevel graph partitioning (METIS) [49], and hierarchical clustering algorithm (GPART) [37]. Previous studies [28,37] have found that in most cases, the combination of the two—a data reordering followed by a computation reordering—gives better results than each alone. In the following discussion, we use *data transformation* to refer to the transformations that use data reordering or/and computation reordering for locality enhancement.

In all prior research, data transformation is applied synchronously with the application. It is placed on the critical path of the application execution, hence subject to the quality-overhead dilemma mentioned in Section 3.1.

3.3 Asynchronous Data Transformation

Asynchronous data transformation is our first technique for resolving the dilemma between transformation quality and overhead. The basic idea is simple: putting data transformation on a helper processor so that it can happen in parallel with the application execution. However, to the best of our knowledge, this simple idea has never been realized before. A plausible reason for the absence is the inherent data dependences between data transformation and the transformed application. To help explanation, we first outline the sketch of a dynamic simulation program, Moldyn, as our example.

```
for each time step
 if time_to_update()
  IList = update_IList (Location);
 end if

 /* main computation with irreg. references to Location */
 for each (i,j) in IList
  f = calculate_force (Location[i], Location[j]);
  Force[i] += f;
  Force[j] -= f;
 end for

 for each particle i
  Location[i] = update_loc (Location[i], Force[i]);
 end for
end for
```

Figure 12: The main loop of Moldyn.

3.3.1 An Example Irregular Dynamic Simulation Program

Moldyn is a program for simulating the movements of many particles caused by their interactions. The program maintains a list, named "interaction list", to record the particles that are close enough to interact with each other. The list consists of a number of pairs; each pair contains the IDs of two particles that are close enough in the particle space to have interactions. Figure 12 shows the pseudo-code of the computation kernel of Moldyn. It contains a time-step loop. In each iteration, the program first checks if it is time to update the interaction list *IList*; if so, it makes the update based on the current locations of the particles. It then traverses the interaction list, and computes the force that a particle receives from its neighbors. After that, the program updates the locations of each particle based on the newly computed forces.

Apparently, the major computation is on the force calculation loop. The references to the *Location* and *Force* arrays in that loop are irregular references; the *IList* array plays the role of an index array, whose content decides which elements of *Location* and *Force* are referenced at which iteration of the loop. Each time when *IList* gets updated, the patterns of the references to *Location* and *Force* change accordingly.

This example shows some representative features of irregular dynamic simulations. These applications usually contain a main loop (e.g., the time-step loop) that encloses irregular references. The irregular references involve two data structures; one is the *reference target* (e.g., *Location* and *Force*), the other is the *reference clue* (e.g., *IList*)[1]. The values of the reference clue often vary across the main loop iterations.

3.3.2 Synchronous Data Transformations

Runtime data transformation can benefit the force calculation loop in the Moldyn example. The basic strategy of a typical transformation is to reorder the items in the reference target according to the content of reference clue—for example, moving particles that have interactions (by reading *IList*) close to one another in *Location* and *Force* in the Moldyn example. Very often, a following computation reordering is applied as part of the data transformation, in which, the iterations of the loop (e.g., the force computation loop) that encloses the irregular references are reordered. For the Moldyn example, it can be realized by reordering the pairs in *IList* based on the new order of particles to further improve the locality.

One place to put the transformation is between the update of the reference clue and the accesses to the reference target, as shown in Figure 13. This placement is natural because of the data dependences among those components. In fact, this placement is what prior studies adopt. Because the data transformation is put on the critical path of the execution, we call it *synchronous data transformation*.

[1]These terms are similar to "index array" and "data array" in some earlier work; using them helps avoid confusion with some other terms in this chapter.

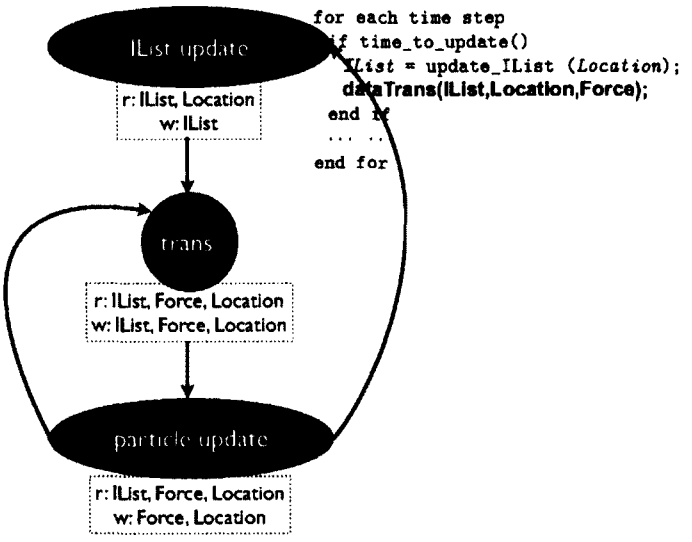


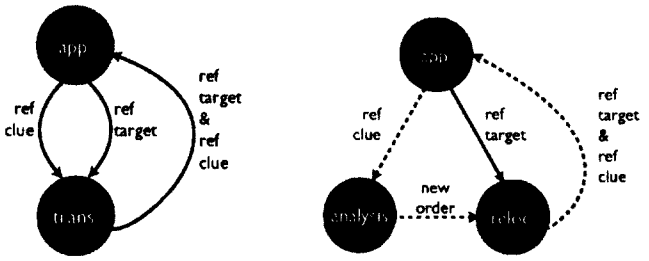
Figure 13: Dependence graph (left) and the synchronous data transformation (right) for the Moldyn example. ("r" and "w" lists the sets of data that are read and written respectively.)

3.3.3 Decomposition and Dependence Relaxation

Data dependences between data transformation and the application form a major obstacle for asynchronous data transformation. Figure 14 (a) summarizes the bi-directional (true) dependences.

We circumvent the dependences based on two properties of data transformations. The first is that most data transformations can be decomposed into an order analysis step and a data relocation step. The order analysis step computes a locality-favorable order according to the reference clue, and the relocation step repositions items in the reference target (and reference clue) based on the produced order. When a data transformation is decomposed into these two components, the two dependence edges from the application to the transformation become pointing to the two components respectively, as Figure 14 (b) shows.

The second is that not all dependences between data transformations and the application are critical. Among the four dependences shown in Figure 14 (b), the dependence from the application to the analysis component is not critical for the correctness of the execution. In another word, if we violate the dependence, the produced locality-favorable order may not lead to a desirable layout for the



(a) Before decomposition (b) After decomposition

Figure 14: Dependences between data transformation and the application. Each edge is a data dependence edge labeled with the related data. Broken edges show dependences that are relaxed in asynchronous data transformation.

reference target, but the execution of the application will be correct still. Similarly, if we violate the dependence from the relocation component to the application, the application may have to use an old layout of the reference target rather than the enhanced one; it may hence run slower than it could, but will still produce the correct result. The same is true for the dependence from the analysis component to the relocation component. On the other hand, the dependence from the application to the relocation component is critical. A violation of this dependence may cause the transformed reference target (e.g., the *Location* and *Force* arrays) to contain obsolete values and impair the correctness of the execution.

Based on the two properties, we develop asynchronous data transformation by relaxing the three non-critical dependences in Figure 14 (b). The key of the implementation is to decompose data transformation into two components, leave the relocation component on the critical path but make the analysis component run by a helper thread asynchronously, and allow the use of obsolete reference clues for the computation of new data orders.

Figure 15 outlines the basic control flow for the Moldyn example. The master thread executes the application and the relocation component, while the helper thread runs the analysis component in parallel. At an update to the interaction list, the master thread sends the new *IList* (or some other reference clue, e.g., coordinates of nodes) to the helper thread, and then continues its execution while the helper thread computes for a new locality-favorable data order. If the new

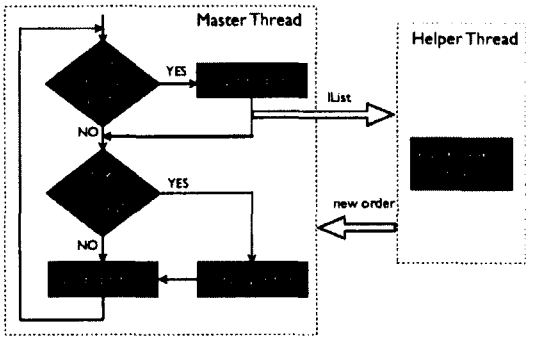


Figure 15: Control flow of asynchronous data transformation for Moldyn.

order is not ready yet when the master thread reaches the "new order ready?" check, it continues executing the following part of the application using current data layouts. When the helper thread finishes computing the order (several time steps may have passed since the order computation starts), it sets a flag so that when the master thread reaches the "new order ready?" check again, it can use the new order to reposition the reference target and reference clue to improve the locality of some following iterations.

This design makes the analysis component of data transformation proceed asynchronously with the application, but leaves the reposition component on the critical path. In many data transformations, the most costly part is in the order analysis rather than the data repositioning–as Figure 19 will show, the time ratios between them are between 6:4 and 8:2 for RCB. This design hence hides the majority of the data transformation overhead. Meanwhile, because the placement of reposition component maintains the critical dependence (the solid line in Figure 14 (b)), the application still runs correctly.

We now examine how the asynchronous data transformation relaxes the three non-critical data dependences (the broken lines in Figure 14 (b)), and the consequences. For the dependence from an application to the analysis component, in the asynchronous transformation, the reference clue passed to the analysis component may be obsolete. It happens when the order analysis takes longer time than an update period of the reference clue. As a result, the new order

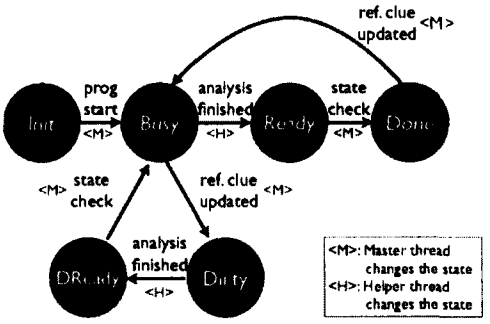


Figure 16: State transitions for thread coordination in asynchronous data transformation.

passed from the analysis component to the repositioning component may be not as good as the computed data order if the current reference clue was used. The ultimate consequence is that the layouts of the reordered reference target and reference clue fit the obsolete rather than the current reference clue well. This analysis reveals a potential loss of the data transformation benefits incurred by the asynchronous scheme. Section 3.6 will show how this loss can be largely prevented.

3.3.4 Thread Coordination

In this part, we present some implementation details on supporting the coordination between a master thread and a helper thread in asynchronous data transformation. The implementation is based on a 6-state transition graph to ensure in-time data transfers and meanwhile avoid unnecessary data copies.

We use a shared variable, protected by a lock, to coordinate the master thread and the helper thread. Figure 16 shows the states recorded by the variable and the state transitions.

When an execution starts and a helper thread is created, the master thread sends the current reference clue to the helper thread, and sets the state to "busy". From the "busy" state, there are two circular paths.

- *Bottom circular path.* When the master thread finishes an update to the reference clue and the state is still "busy", it changes the state to "dirty", indicat-

ing that a new order needs to be computed because the reference clue has changed. When the helper thread finishes its current job and passes its computed order to the master thread, it changes the state from "dirty" to "dready". At the next state check by the master thread, it will see that the helper thread has just prepared a new data order and also it needs to get the current reference clue to compute another data order. The master thread then sends the current reference clue to the helper thread, conducts a reposition transformation using the new order, and then changes the state to "busy".

- *Right circular path.* If the helper thread finishes its job within one update period of the reference clue, it changes the state to "ready". At the next state check by the master thread, it will see that the helper thread has just prepared a new data order. It conducts the reposition transformation and then changes the state to "done". Note that it does not send the current reference clue to the helper thread because the clue is identical to what the helper thread already has, which is the key difference between the "read" and "dready" states. At the next update of the reference clue, the master thread sends the clue to the helper thread and changes the state to "busy".

The design of the state transitions ensures that both threads receive necessary data in time, and meanwhile avoids unnecessary data transportation. For instance, consider a case where during the computation of one new data order, the master thread updates the reference clue three times. The state will remain "dirty" after the first update until the helper thread finishes its job. As the master thread sends no data in the "dirty" state, only the most recent reference clue (i.e. the one after the third updates) is sent to the helper thread.

Asynchronous data transformation hides most transformation overhead, but is subject to the use of obsolete reference clue. The longer a transformation takes, the more obsolete the used reference clue is. Even though in many cases, the

gain exceeds the loss as Section 3.7 will show, reduction of the transformation time will make its benefit more pronounced—the goal of the technique presented next.

3.4 TLayout: A Transformation Algorithm for Throughput-Oriented Processors

The second technique we develop is *TLayout*, a data transformation algorithm for reducing transformation overhead by exploiting the special features of throughput-oriented processors.

The motivation comes from the trend in modern architecture development. Due to the high throughput and power efficiency, throughput-oriented processors (e.g., GPU) are being increasingly adopted to co-run with general-purpose CPUs. This trend is underscored by the recent Intel Sandy Bridge and upcoming AMD Fusion processors, which have CPU and GPU on a single chip. Exploiting throughput-oriented co-processors for irregular applications is a challenge, given that these massively parallel co-processors are typically weak in handling computations with complex memory references, dependences, and control flows.

Our idea is to use such co-processors to accelerate data transformations for CPU executions. This use of the co-processors is especially appealing for legacy CPU code, because it needs virtually no code changes. Programmers only need to insert three function calls (see Section 3.5) to invoke a data transformation function we have developed for the co-processors. In contrast, many efforts are needed for porting and tuning an irregular application to co-processors [99, 100]. The paradigm of using co-processors for program optimizations offers an *easy, quick* way for legacy programs to benefit from the co-processors (even though the performance from manual code porting may be higher).

Unfortunately, none of previous data transformation algorithms is designed for massively parallel architectures. Their complex control flows and dependences make them unsuitable for throughput-oriented processors.

TLayout Algorithm *TLayout* is our solution to the problem. It is designed to be massively data parallel. It produces locality of the similar quality as sophisticated classic transformation algorithms do, but with one third of the overhead (on GPU).

As with many previous data transformation algorithms [37], TLayout is based on the underlying graph structure of data references in the application. Simply speaking, data elements that are referenced closely (e.g., in one iteration of an inner loop) are regarded as neighbor nodes in a reference graph, having an edge in between. Data locality optimizations are then mapped to a graph partitioning problem. Partitioning the graph and putting nodes in a partition close in memory usually improves spatial and temporal locality. In dynamic simulation programs, the reference graphs are often already embedded in the reference clue—such as the interaction list in Moldyn, and the mesh structure in a mesh refinement application. As the reference graphs of these applications usually come from the spatial or topological relations among objects (e.g., particles in a physical space), it is typical that one reference graph covers all interesting data objects.

The strategy of TLayout is incremental clustering through iterative membership-propagation based on the topology of the reference graph. The input to TLayout is a reference graph, encoded as a number of node pairs, with each pair consisting of two nodes that are connected by an edge in the reference graph. The output is a number of clusters that partition the nodes of the graph completely in a way that the nodes close in topology belong to the same cluster. The algorithm starts by setting the membership of each node (i.e., which cluster it belongs to) to *null*. It then proceeds in the following steps:

1. SEED PLANTING: TLayout randomly selects K nodes as the seeds for K clusters.

2. PROPAGATION: Every node whose membership is *null* checks the membership of its neighbors one after one. As soon as it encounters a neighbor whose membership is not *null*, this node changes its own membership from *null* to the membership of that neighbor.

3. LOOP: Repeat STEP 2 until the fraction of nodes having *null* membership is below a preset threshold δ, or the number of times the propagation step has been invoked reaches a preset upperbound U.

4. (optional) HIERARCHY CONSTRUCTION: Recursively merging the clusters based on their closeness on the reference graph into a hierarchy.

5. LAYOUT: Finally, arrange nodes according to the resulting clusters. Nodes in the same cluster are laid out nearby in memory. If a cluster hierarchy is created, the leaf clusters are processed following their appearance order in the hierarchy.

The propagation step dominates the time cost of the algorithm. But it is a completely data-parallel process, meeting the strength of throughput-oriented processors.

Parameters and Adaptive Control There are two parameters in the TLayout algorithm. The use of a small positive value of the parameter $delta$ allows the algorithm to stop with a small portion of nodes carrying *null* membership. These nodes will be attached to the end of the final data layout. As the number is small, they have little influence on the quality of the resulting data layout. But using such a value may save one or multiple invocations of the propagation step. Like many parameters used in practical systems, users set this value based on their experiences and preferences. We use 1% as its value for all our experiments.

The second parameter is the number of clusters K. A large value of K leads to quick membership propagation, hence few invocations of the propagation step. However, it may hurt the quality of the resulting data layout: Many nodes that have good reference affinity may fall into different clusters. The optimal value of K depends on the graph properties and the application. As the reference graph periodically changes throughout the execution of a dynamic simulation program, its value is difficulty for users to set.

We design an adaptive control to automatically adjust the value of K. After each data transformation, TLayout compares the transformation time and the length of the update period of the reference clue. If the transformation time is too long, TLayout doubles the value of K to accelerate the next data transformation. Typically, K starts with a small value (100 in all our experiments).

Implementation on GPU TLayout is designed for general massively parallel architecture. We implement it using CUDA [3] in machines equipped with GPU. CUDA is a C-like interface for GPU programming. A CUDA program consists of a CPU code and a GPU code. The code executed on GPU is wrapped in functions called GPU kernels. A GPU typically contains hundreds of cores. There is a certain amount of on-chip memory (called shared memory) and a large chunk of off-chip memory (called global memory). When a GPU kernel is invoked, hundreds of GPU threads are launched to run the same GPU kernel with the same parameters. Each thread has one unique ID number; the kernel may use thread ID to trigger different behaviors of different threads.

In our implementation, each GPU thread manages one node in the graph. Algorithms 2 and 3 outline the CPU code and GPU kernel respectively.

Discussions TLayout has some appealing characteristics worth mentioning. First, it well exploits the massive parallelism of GPU. Assigning a thread to every node makes the algorithm simple to implement and proceed efficiently. Second,

Algorithm 2 *TLayout(num_nodes, num_edges, neighbor_list)*

```
1: // build a single array to store neighborhood info to prepare for GPU kernel
   execution
2: for i = 0 to num_edges − 1 do
3:    left = neighbor_list[i][0];
4:    right = neighbor_list[i][1];
5:    if neighbor_size[left] < MAX_NB_Per_Node then
6:       neighbors[neighbor_size[left] + +] = right;
7:    if neighbor_size[right] < MAX_NB_Per_Node then
8:       neighbors[neighbor_size[right] + +] = left;
9: //randomly select K nodes as the seeds for K clusters
10: for i = 0 to K − 1 do
11:    membership[rand()%(num_nodes)] = i;
12: while too many nodes have null membership do
13:    //invoke GPU kernel for neighborhood-based clustering
14:    TLayoutKernel<<< ... >>>(neighbors, membership);
15: //merge clustering results to generate a new data order
16: for i = 0 to num_nodes − 1 do
17:    id_cluster = membership[i];
18:    cluster_lists[id_cluster].append(i);
19: ind = merge_lists(cluster_lists);
20: return ind
```

a node having a high degree tends to grab more nodes into its cluster than other nodes do, which is a desirable property for spatial locality. Third, the algorithm adaptively selects the appropriate number of clusters. This adaptivity fits the dynamic properties of irregular simulation well. We note that TLayout specifically exploits the massive parallelism in throughput-oriented devices (e.g., GPU). It is not intended to be used on CPU. (Experiments show it is tens of times slower than RCB on a CPU.)

3.5 Asynchronous Data Transformation Library (ATrans)

We integrate the techniques, along with previous transformation techniques, into a Asynchronous Data Layout Transformation library (ATrans) to simplify their use. ATrans consists of all the support for asynchronous transformation, the adaptive TLayout algorithm, and a set of previously implemented data transformation func-

Algorithm 3 *TLayoutKernel(neighbors, membership)*

```
1: // load neighbors into shared memory
2: . . .
3: // membership propagation
4: i = global_thread_number;
5: for j = 0 to MAX_NB_Per_Node − 1 do
6:   neighbor = get_neighbor();
7:   if membership[neighbor] & !membership[i] then
8:     //propagate membership
9:     membership[i] = membership[neighbor];
10:    break;
```

tions from University of Maryland [37]. It supports the asynchronous data transformation on both CPU and GPU.

Its usage is simple. To enable asynchronous data transformation for an application, it typically requires just an insertion of three function calls in the application program, one in the initialization stage, one after the update of the reference clue, and one at the beginning of the central loop (e.g., the time-step loop in Moldyn). Figure 17 illustrates the use of the library for Moldyn. The *ATrans_init_pipeline* function indicates whether CPU or GPU is to be used for analysis component, creates a helper thread, initializes the state of the pipeline and necessary data structures, and prepares the GPU execution if GPU is used. When the interaction list is updated, the *ATrans_analysis* function checks the pipeline state and wakes helper thread up to do transformation analysis if necessary. At the beginning of each iteration of the time-step loop, the *ATrans_reposition* function checks the state and reposition the data if it is time to do so.

3.6 Adapting On The Fly

The benefits of asynchronous transformation do not come for free. Recall that to circumvent the data dependence, it uses obsolete reference clues as heuristics for data transformations. Although in many cases the benefits outweigh the catch, it is not always so. Whether an asynchronous transformation excels a synchronous

```
int main (int argc, char **argv)
{
  /***initialization of the simulation***/
  ATrans_init_pipeline(__ATRANS_CPU, interaction_list,
    MAX_EDGE, coordinates, MAX_NODE, forces);
  for(iter = 0; iter < NUM_ITER; iter++)
  {
   if(update_interaction_list() == true)
    ATrans_analysis();
   ATrans_reposition();
   /***simulation kernel code***/
  }
  /***deal with result***/
}
```

Figure 17: Use of the ATrans library in Moldyn. Inserted codes are function calls with prefix "ATrans_".

transformation is subject to the ratio between transformation overhead and per iteration computation time, the frequency of the update to reference clues, the speedup, and so on.

We devise an online adaptive scheme to select the suitable transformation strategy on the fly. The basic idea is to estimate the benefits of different strategies during the initial time steps, and then apply that strategy to the remaining time steps.

To figure out the overall benefits of synchronous transformation, it is necessary to determine the best frequency to apply it. Because of its transformation overhead, applying it at every update of reference clues is often sub-optimal. We employ a prior method [61] to solve the problem. By applying the synchronous transformation only once, it can determine the best frequency and estimate the overall benefits by observing the computation speed in a number of iterations following the transformation. The process introduces no extra overhead.

Figuring out the overall benefits of asynchronous transformation is less straightforward. As asynchronous transformation is off the critical path, it can be applied often. In our design, it is applied at the reference clue update following the finish of the previous asynchronous transformation. Because the per iteration time

varies across update periods, it is difficult to get a closed form to compute all the ending/starting time points of asynchronous transformations, causing difficulty for benefit estimation.

Our solution is to emulate the timeline of the kernel computation and the applications of asynchronous transformations. For space constraint, we describe it briefly. It requires the following parameters: the analysis and reposition times of a data transformation, the frequency of reference clue update, the total time steps of the kernel computation, and computation speed in a number of iterations following the transformation. Attainment of these numbers needs one application of the transformation only. An emulation of the timeline involves the computation of a number of linear expressions for calculating when each asynchronous transformation will apply and how many time steps of computation can benefit from it. The emulation takes less than 0.01% of overall running times in all our experiments.

After estimation of the overall benefits of synchronous and asynchronous transformations, the winner will serve for the rest of the execution. Although the program may not be using the optimal transformation scheme during the initial time steps, the next section will show that the influence is small as these steps take only a small portion of the entire simulation.

The adaptive scheme may suffer if some key factors (e.g., frequency of reference clue update) change dramatically across time steps. Fortunately, most dynamic simulations do not see such drastic changes.

3.7 Evaluation

We conduct a series of comparisons to evaluate the values of the techniques. We give an overview of the results first.

- *Asynchronous Transformation:* As Section 3.3 mentions, asynchronous transformation hides most overhead, but its use of obsolete reference clues may

have certain side effects on the resulting locality. We conduct a head-to-head comparison between asynchronous and synchronous transformations (both using RCB on CPU) in terms of the overall performance and resulting locality of the transformed applications. The result shows both the strength and weakness of the asynchronous transformations. On three benchmarks, asynchronous transformations lead to 18% more speedup than synchronous transformations do. In addition, for these benchmarks, using an extra CPU core for transformation brings 15% more speedup than using that core for computation, justifying the resource usage of the asynchronous transformations. On the other hand, the negative effects of obsolete reference clues outweigh the benefits of asynchronous transformations on two other benchmarks, leading to slightly less speedup than the synchronous transformations do.

- *Runtime Adaptation:* The runtime adaption scheme is able to identify the best transformation strategy for all benchmarks. With low overhead, it helps exert the strength of both asynchronous and synchronous transformations.

- *TLayout:* By comparing with a prior sophisticated algorithm (RCB), we observe that in most cases, the TLayout algorithm produces data layout of similar quality as the prior algorithm does, but takes around one third time to run.

- *Overall:* When the techniques are applied together, they generate 1.3–3.1X speedup over the original performance of five benchmarks, outperforming the state-of-the-art data transformation techniques significantly.

The conclusions obtained are based on measured wall-clock times and confirmed by hardware performance counters results.

3.7.1 Methodology

Platform All experiments happen on a dual-socket dual-core AMD Opteron 2216 machine in the National Center for Supercomputing Applications. The machine

is equipped with an NVIDIA Tesla S1070 GPU with 16GB DDR3 memory. It consists of four Tesla T10 C1060 GPUs, with each containing 240 cores, organized in 30 streaming multiprocessors. We use only one of the GPUs in our experiment. The machine runs Linux 2.6.33. We use GCC 4.3.2 (with "-O3" flag) as the compiler and CUDA 3.0 as the GPU programming model. We employ libpfm4 [1] for collecting cache performance data.

Benchmarks We concentrate our experiments on a dynamic simulation benchmark suite from Han and Tseng [37], and two other programs, Mesh and CFD, respectively from the Chaos group [25, 103] and the Fluid Dynamics community [21]. The suite from Han and Tseng consists of three representative programs, Nbf, Irreg, and Moldyn. They are all derived from real applications. Nbf is abstracted from GROMOS, a force field of molecular dynamics simulation; Irreg is the kernel of an iterative partial differential equation solver; Moldyn is from a molecular dynamics simulation named CHARMM. These three benchmarks have been commonly perceived to be representative, and have served as the *only* benchmarks in some influential data locality studies in dynamic simulations [37, 52, 85]. We add two more benchmarks to increase the coverage. Mesh is an unstructured mesh simulation. CFD is an unstructured grid finite volume solver for three-dimensional Euler equations for compressible flow. Similar to the extensions Han and Tseng made to Irreg [37], both Mesh and CFD are modified to accommodate dynamic changes in the underlying mesh or grid structures.

Table 3 lists the properties of the inputs used with these benchmarks. FOIL and AUTO are 3D meshes of a parafoil and GM Saturn automobile, respectively. MOL1 and MOL2 are small and large 3D molecule models originally obtained from MOLDYN application. The results on all inputs show similar performance trends. Due to space constraints, our discussion concentrates on the results on large inputs (AUTO and MOL2) for the severity of their locality issues.

The frequency of the update to reference clues affects the problem setting and the potential of runtime data transformations. We experiment three typical frequencies: one update in every 10, 20, or 30 iterations of the main computation loop of the applications.

Transformation Frequency Data transformation can be applied as often as once per update of the reference clue, or once every several updates. The more frequent it is applied, the better the locality of the application is, but meanwhile, the more overhead it incurs.

For the asynchronous paradigm, the transformation frequency is automatically determined by the 6-state master-helper coordination scheme as described in Section 3.3.4. The problem is tricky for synchronous transformations. For a fair comparison, one seemingly straightforward option is to use the same transformation frequency as the asynchronous transformation uses. But this option is in fact unfair to the synchronous scheme. That frequency often causes much worse performance than some other frequencies for synchronous transformations. A previous study [61] introduces a method to analytically determine the optimal frequency for synchronous transformations. We have verified the optimality of the method through a sequence of empirical measurements. In all our experiments, we use optimal frequencies found in that way for synchronous transformations.

Algorithm and Others We select previously proposed RCB algorithm (implemented by Han and Tseng [37]) as the analysis algorithm in all CPU experiments, synchronous or asynchronous. RCB has been shown to be one of the most sophisticated algorithms that produce the largest locality enhancement for most benchmarks [37]. Our experiments echo that despite it is more expensive than some other methods (e.g., CPACK), its overall performance is often among the best when it is applied synchronously at the best frequency.

Table 3: Inputs*

Name	# Nodes	# Edges	Description
FOIL	144649	1074393	3D mesh of a parafoil
AUTO	448695	3314611	3D mesh of GM's Saturn
MOL1	131072	1179648	3D molecule distribution (sm)
MOL2	442368	3981312	3D molecule distribution (lg)

*: come from Han and Tseng [37].

As the asynchronous transformation is mainly on data reordering, we apply the same computation reordering (lexicographical sort) to all experiments. The computation reordering overhead is small (less than one seventh of RCB) and is counted in data repositioning overhead in all experiments.

3.7.2 Experimental Results

We experiment both single-thread and parallel executions of the benchmarks. They show similar conclusions. We first give a detailed analysis using the single-thread results, and then report the parallel results at the end, along with the justification of the resource usage by asynchronous transformations.

Sequential Executions

Figure 18 shows the comparison of overall running times. Each time consists of the application running time and all transformation overhead that is not hidden (including data transfer between CPU and GPU).

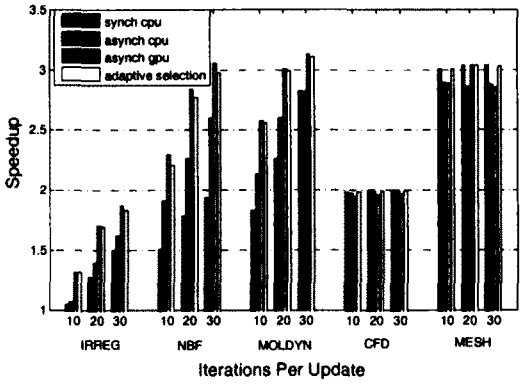


Figure 18: Speedup of the overall executions for single-threaded benchmarks. The speedups are over single-threaded benchmarks without any data transformation applied.

IRREG, NBF, MOLDYN On the first three benchmarks, the synchronous transformations show 77% average speedup. The asynchronous transformations on CPU show 18% more average speedup. The benefits come from two aspects. First, the asynchronous scheme hides significant transformation overhead, as Figure 19 reports. The second benefit relates with the first. Because the transformation incurs smaller overhead on the critical path than the synchronous scheme does, it is automatically applied more frequently by the master-helper coordination scheme than the synchronous one. The more frequent transformation yields better locality, confirmed by the L2 cache miss rates shown in Figure 21. The figure shows a few exceptional cases (e.g., the configuration "Moldyn 30"), in which, the two transformations are applied at the similar frequencies; the use of obsolete reference clues causes the relatively less locality enhancement. However, thanks to the overhead hiding by the asynchronous transformation, it leads to the better or similar overall performance still, as Figure 18 shows.

We stress that the synchronous results are what we get when the optimal transformation frequency is used. Increasing invocation frequency of the synchronous transformations yields only worse overall performance due to the large overhead incurred, while decreasing the frequency worsens the performance as well due to the less locality enhancement to the application, as Figure 20 illustrated.

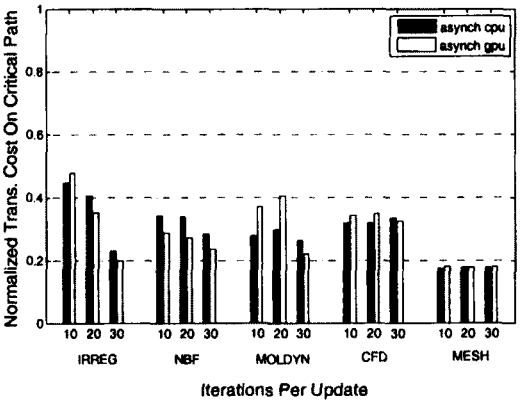


Figure 19: Optimization cost on critical path. The results are normalized over those of synchronous transformation.

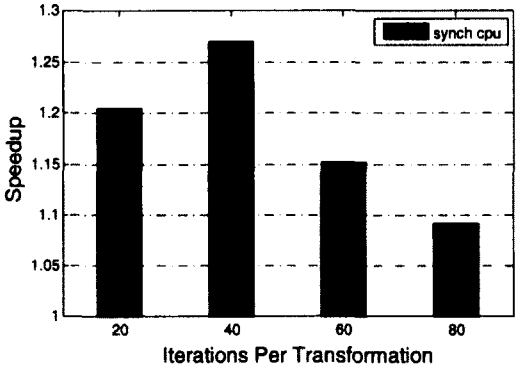


Figure 20: Speedup of IRREG with different transformation frequencies. Neighbor list is updated every 20 iterations.

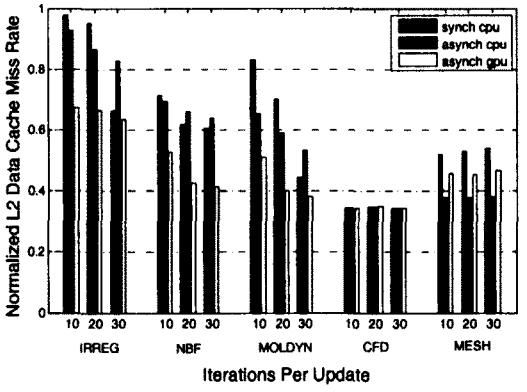


Figure 21: L2 cache performance comparison between synchronous and asynchronous data transformation. Results are normalized over those without any transformation.

The asynchronous TLayout produces even larger benefits than the asynchronous CPU approach. The extra speedup ranges from 28% to 112% with an average of 65% over those of the synchronous scheme, and 25–58% better than those of the asynchronous CPU results. The extra benefits come from two appealing features of the TLayout algorithm. First, it runs 2.8 to 3.3 times faster than the RCB algorithm, thanks to its effective exploitation of the throughput-oriented processors. Second, it produces data layout of comparable quality as the sophisticated RCB algorithm does as Figure 22 reports. These two features together explain why asynchronous TLayout produces better data locality than the asynchronous CPU does. The second feature ensures that each invocation of the data transformation in the two schemes are similarly powerful, while the

first feature entails much more affordable invocations of data transformations in the asynchronous TLayout than in the asynchronous CPU.

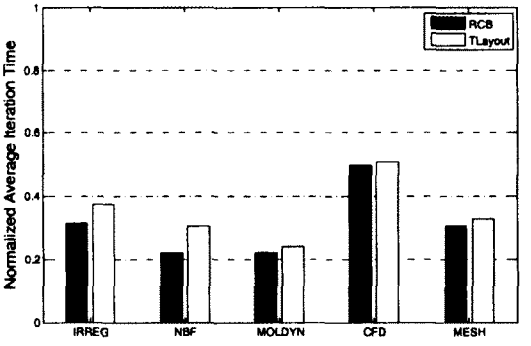


Figure 22: The time per iteration of the computation loop after a transformation is applied. It is the average of 100 iterations following the transformation. The results are normalized over those of single-threaded benchmarks with no transformations applied.

CFD and MESH The results on these two programs show a trend different from the other three programs. On both of them, changes to the reference clue during the simulations are less significant than on the other programs. The overall speedups from the transformations are still large because the initial data layout is inferior. However, because the changes are small during the simulation, there is no need to apply transformations often. There is limited overhead for asynchronous transformations to hide. Consequently, the negative effects of the use of obsolete reference clues become noticeable. So on both programs, regardless the reference clue update frequencies, the asynchronous transformations perform slightly worse than the synchronous transformation. The L2 cache results of Mesh in Figure 21 seem counter-intuitive: Asynchronous ones are lower than the synchronous one. A plausible reason is that the locality of the program is mainly embodied by other metrics. For instance, the synchronous scheme has L1 cache miss rate half of that of the asynchronous GPU scheme.

Adaptive Selection The adaptive selection scheme successfully selects the best strategy to use for all cases. Because some transformations in the initial time steps do not use the optimal strategy, there are slight differences between

the speedups from the adaptive scheme and those of the best strategy. However, overall, it achieves the near best performance on all benchmarks, showing the promise for exerting the strength of both asynchronous and synchronous transformations.

Parallel Executions

Figure 23 reports the similar comparison but on parallel executions of the benchmarks. For the baseline (i.e. no transformations applied) and "synchronous CPU", we use 4 threads for each benchmark as the machine contains 4 cores. In "asynchronous CPU" and "asynchronous GPU" case, we use 3 threads for each benchmark so that the transformation can happen on the remaining core.

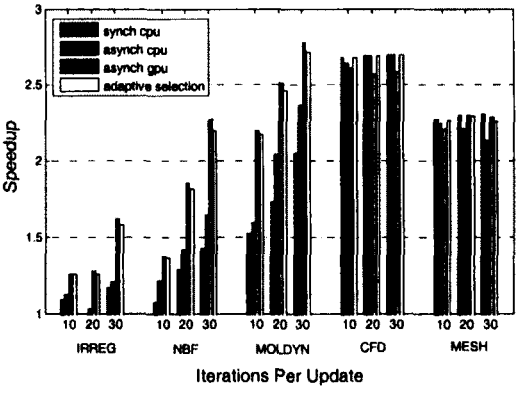


Figure 23: Speedup of the overall executions for parallelized benchmarks. The speedups are over parallelized benchmarks without any transformation.

The results show similar conclusions as the single-thread experiments do. One particular point we want to mention is that even though the "asynchronous CPU" uses one fewer worker threads than "synchronous CPU", with the help from the asynchronous transformation, it still excels in resulting performance. Part of the reason is that the irregular applications have many communications among threads due to the inherent properties of the applications. As a result, the parallel program shows sub-linear performance scalability in the number of threads. Adding the fourth worker thread improves the performance of the programs by 6%, exceeded by the benefits from the asynchronous transformations. The results

justify the resource usage of the asynchronous transformations. In addition, the parallelization imposes different influence on the locality of different benchmarks. The locality issue of CFD becomes especially serious after the parallelization, hence the large benefits from data transformations.

More Results on TLayout Algorithm

The speed of membership propagation in the algorithm determines the number of iterations the propagation has to happen (to reach the predefined threshold δ). In all the experiments reported in previous sub-sections, the average numbers of needed propagation iterations are no larger than four. This result indicates the high speed of membership propagation. Analytically, it may be attempting to think that if the closest center is K hops away from a node, it would take K iterations of propagation for that node to be clustered. However, because global memory is used for membership labels, during an iteration of propagation, the membership of a node becomes visible to all threads (e.g., all nodes) immediately after the node gains its membership. For instance, in Figure 24, node N_3 can be clustered in one propagation if either of the following two conditions is met: (1) N_3 is visited after N_2 and N_2 is visited after N_1; (2) N_3 is visited after N_5 and N_5 is visited after N_4. In TLayout, the visiting order of nodes is random; in the GPU implementation, the order is determined by the scheduling of GPU threads, which exhibits large randomness.

To examine the scalability of the algorithm, we create a spectrum of problems of different sizes. At each size, we run the algorithm 7 times to get the average number of propagations required to cluster 99% nodes. As the focus is on assessing the propagation speed, we fix the number of clusters to be 100 in all runs. Results in Figure 25 demonstrates the good scalability of the algorithm.

Overall, the results demonstrate that both the asynchronous data transformation and the TLayout algorithm are able to produce certain degrees of benefits for

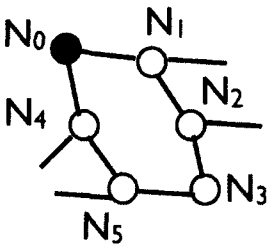


Figure 24: An example showing the membership propagation in TLayout. The filled node is already clustered; the others are not.

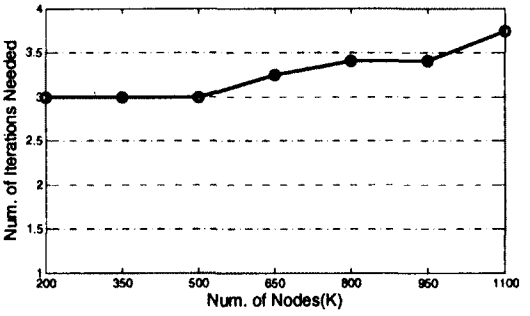


Figure 25: Scalability of TLayout

the enhancement of data locality of irregular dynamic simulations. Together with the online adaption scheme, they resolve the quality-dilemma faced by existing data transformation techniques, and yield significant performance improvement.

3.8 Related Work

In Section 3.2, we reviewed some prior data reordering and computation reordering techniques for the enhancement of data locality of dynamic simulation programs. In addition, Strout and others have proposed a compile-time framework that allows the explicit composition of run-time data and iteration reordering transformations [85]. Kulkarni and others [53] have studied locality issues of irregular applications in the context of optimistic parallelism. They concentrate on the partition of data among threads rather than data layout reorganizations for locality improvement.

Recent years have seen a rapid increase of the use of GPU for data-parallel computing. Previous work on CPU-GPU cooperative computing concentrates on

offloading some computation-intensive and easily parallelizable parts of an application to GPU. In this scenario, the key issue is how to partition the jobs among GPU and CPU [76], and how to optimize GPU code to maximize the computing efficiency on GPU through compiler techniques [9, 56, 97], runtime optimizations [99, 100], or empirical search-based optimizations [59]. Some recent studies attemp to enable seamless translation between GPU and CPU code [33, 34, 84]. We are not aware of prior proposals in using GPU to do runtime optimizations for CPU computing.

There are many clustering algorithms developed in the machine learning area [39]. But most of them are distance-based (e.g. K-Means) rather than topology-based. Our search yields no satisfied topology-based clustering algorithm that is simple and fits GPU well, hence our development of TLayout.

3.9 Summary

This chapter presents three techniques for resolving the quality-overhead dilemma of data transformations for irregular references. The first, *asynchronous data transformation*, moves data reordering off the critical path through dependence circumvention and layout approximation. The second, *TLayout*, is a novel data transformation algorithm designed to take advantage of modern throughput-oriented processors. The third technique, *adaptive control*, allows transparent selection of suitable transformation schemes for an execution. Together, they improve the performance of some irregular dynamic simulations significantly. In addition, this study initiates a new way of collaborations between CPU and co-processors, which may lead to some unconventional directions for program optimizations in a heterogeneous computing environment.

Chapter 4

Enabling Program-Level Control of Scheduling on GPU

4.1 Introduction

With hundreds of cores integrated, GPU often creates tens of thousands of threads for an application. The massive parallelism produces large potential throughput, but also imposes grand challenges for thread management, or scheduling.

Scheduling determines when and where a task is processed. It is essential for matching communication and memory access patterns with underlying architecture, in order to fully tap into the power of a parallel system. Scheduling is usually controlled by thread schedulers. On CPU, the thread scheduling is implemented through system APIs. But on GPU, there is no such software API; the scheduling on GPU has been controlled by hardware and runtime. Such a design is demanded by the scale of parallelism: Hundreds of thousands of threads need to be scheduled in no time. However, the lack of software-level control of scheduling forms a major barrier for software to leverage scheduling to optimize program executions. What increases the barrier is that the scheduling algorithms employed by GPU hardware and runtime have remained non-disclosed;

the schedulers vary substantially across generations and have exhibited some obscure and non-deterministic behaviors (detailed in the next section).

The restrictions have drawn some recent attentions from researchers in various domains. A number of studies independently invented the method of *persistent threads* to go around the hardware scheduling problem [6, 19, 35, 89, 95]. The idea is to create only a small number of threads that can simultaneously run actively on a GPU. Unlike in traditional kernels where a thread terminates as it finishes a task, these threads stay alive throughout the execution of a kernel function. They continuously fetch and execute tasks from one or more task queues. By controlling the order of the tasks in the queues, one can match the executions with some communication patterns among tasks—for example, putting a producer and its consumer into the same queue. thread.

Although persistent threads offers some support to task scheduling on GPU, the support is restrictive. It only decides which tasks map to which persistent thread and their execution order; it gives no support for deciding *where or on which processor a task should run*. Such location control is still up to the hardware and proprietary runtime, which decide the placement of persistent threads, and hence the placement of tasks associated with those threads.

Lack of such scheduling control at the spatial dimension hinders persistent threads in supporting optimizations that are related with non-uniformity in processors. For instance, a modern GPU consists of multiple streaming multiprocessors (SM), with each containing tens of cores. Cores on one SM usually share some on-chip storage on that SM (e.g., L1 cache and texture cache). As a result, one task may be able to read the data in a cache brought by another task that concurrently runs on the same SM. With location control, one could make two tasks that share lots of data run concurrently on the same SM[1]. Such optimizations are especially beneficial for tasks with non-uniform data sharing, which include tasks

[1]Mapping two tasks to the same persistent thread can also make them map to the same SM, but the tasks have to run serially by that thread, throttling the benefits of synergistic data fetching.

of many irregular applications (e.g., N-body simulations), as well as tasks coming from different kernels (or applications) that are deployed concurrently on a GPU. Besides for data sharing, the spatial control is critical when there are architectural variations among SMs. Unintentional variations among SMs in a GPU already widely exist today [40]; with frequency scaling [54] possibly implemented in future GPU, even more substantial (intentional) variations (e.g., different SMs could be reconfigured to different clock frequencies to balance energy and performance) are possible. In these scenarios, spatial control of scheduling is important for matching tasks with the suitable SMs.

In this work, we show that spatial scheduling control actually can be enabled through a simple program transformation, called *SM-centric transformation*.

SM-centric transformation includes two essential techniques. The first is *SM-based task selection*. In a traditional GPU kernel execution, with or without persistent threads, what tasks a thread executes are usually based on the ID of the thread (or determined randomly in a dynamic task management). While with SM-based task selection, what tasks a thread executes is based on the ID of the SM that the thread runs on. By replacing the binding between tasks and threads with the binding between tasks and SMs, the scheme enables a direct, precise control of task placement on SM.

The second technique is *filling-retreating scheme*, which offers a flexible control of the amount of active threads on an SM. Importantly, the control is resilient to the randomness and obscuration in GPU hardware thread scheduling. It helps SM-centric transformation in two aspects. First, it ensures an even distribution of active threads on SMs, which is vital for guaranteeing the correctness of SM-centric transformations. Second, it facilitates online determination of the parallelism level suitable for a kernel, which is especially important for the performance of multiple-kernel co-runs, a scenario benefiting significantly from SM-centric transformation.

SM-centric transformation, by enabling flexible program-level control of task scheduling, opens up many new opportunities for optimizations. In our experiments on 72 co-runs of kernels, it helps produce on average 33% improvement in system throughput and turnaround time. When applied to locality enhancement, the enabled spatial scheduling shortens the execution times of four irregular applications by 20% on average. In both cases, it significantly outperforms the support that persistent threads provide. These results indicate that SM-centric transformation, by complementing prior methods, provides a critical missing piece of the puzzle for enabling a flexible control of task scheduling on GPU.

4.2 Background

We base our discussions on terms in NVIDIA CUDA [2]; but the technique could be applied to other GPU programming models.

Organization of Cores and Threads As a massively parallel architecture, a GPU consists of a number of *streaming multiprocessors (SM)*, with each containing tens of cores. A GPU usually creates a large number of threads at the launch of a *kernel* (i.e., a CPU-invoked function that runs on GPU). These threads are typically organized in a hierarchy: 32 compose a *warp*, many warps compose a *thread block* or called a *cooperative thread array (CTA)* and many CTAs compose a *grid*.

Spatial Scheduling A CTA is the unit for spatial scheduling: At a kernel launch, the GPU hardware scheduler named GigaThread [70] assigns each CTA to one of its SMs. The assignment algorithm has not been disclosed to the public. It differs from one generation of GPU to another, and exhibits lots of irregularity. For example, our experiments on Tesla M2075, a type of widely used workstation GPU, show different CTA-to-SM assignments in two repeated invocations of the

same kernel on the same input, and neither is in a round-robin or other regular predictable pattern.

Temporal Scheduling A warp is the unit for temporal scheduling: All threads in a warp proceed in lockstep. Many CTAs may be assigned to an SM, but at one time point, only a limited number of them can be active—meaning that they attain enough registers and other hardware resources and are ready to run. All other CTAs have to wait until some active CTA finishes executing the entire kernel function and releases some hardware resources.

Non-Uniformity on GPU Spatial scheduling is potentially beneficial to GPU, as non-uniformity exists on both GPU resource sharing and its workload.

On the resource sharing aspect, modern GPU features non-uniform cache sharing. In Tesla M2075, for instance, there are 14 SMs, with each containing some cache—such as, instruction cache, L1 data cache, constant cache, and texture cache—that is shared by all cores on that SM but is not accessible by other SMs.

On the workload aspect, non-uniformity shows in two levels. For a single GPU kernel, a CTA may share different amounts of data with different CTAs. Molecular Dynamics (MD) simulation is such an example. It simulates interactions among neighbor atoms. The atoms simulated by two CTAs may have many or few neighbors, depending on the distances among them in the simulated space. That naturally leads to non-uniform data sharing among CTAs. Meanwhile, recent generations of GPU start to support concurrent executions of multiple kernels on a single GPU. Although currently the kernels have to be launched from a single CUDA context, a more general support for concurrent executions of multiple GPU applications is expected to come in the near future. In these co-run scenario's, non-uniformity becomes even more common: CTAs from the same kernel often share more instructions and data than CTAs from different kernels do.

```
kernel_org()
  jobID = f (workerID);
  processJob (jobID);
```

(a) original kernel

```
createJobQ4workers(); // on CPU

kernel_persist()
  while (jobID = JobQ[workerID].next() != null)
    processJob (jobID);
```

(b) with persistent thread

```
createJobQ4sms(); // on CPU

kernel_smc()
  smID = getSMID();
  jobID = JobQ[smID].next();
  if (jobID != null)
    processJob (jobID);
```

(c) with SM-centric task selection

Figure 26: Conceptual relations among jobs, workers, and SMs.

The non-uniformity suggests the potential of spatial scheduling. As Section 4.6 quantitatively confirms, a good spatial scheduling may bring an over 30% speedup on average.

4.3 SM-Centric Transformation

At the center of SM-centric transformation are two techniques: SM-centric task selection, and a filling-retreating scheme. In this section, we first explain the basic ideas of the two techniques and how they complement each other to form a single solution to circumvent the limitation from the hardware scheduler. As the techniques are generally applicable to various GPU programming models, we use high-level pseudo-code for description and skip detailed complexities in implementation so that the ideas can be easily grasped by general readers. We then use CUDA as an example programming model to explain the detailed implementation of the techniques, including some subtle considerations that are critical for the techniques to work efficiently. We show that the entire SM-centric transformation can be conducted through a simple pass by compilers. At the end, we give some discussions on the soundness of the transformation and its applicable conditions.

4.3.1 SM-Centric Task Selection

Basic Idea SM-centric task selection associates tasks with SMs. We explain it based on the following abstract model of GPU kernel executions.

Commonly, an invocation of a GPU kernel causes many GPU threads to create, which are often organized in a hierarchical structure. At an abstract level, the execution of a GPU kernel can be regarded as consisting of many jobs[2] conducted in parallel by a number of workers on some SMs. Here, a worker corresponds to a group of GPU threads (e.g., a CTA), and a job corresponds to the operations conducted by such a thread group, including all their data accesses. There is a unique ID number associated with each job, worker, and SM.

In traditional GPU programs, which job a worker does has been determined by the worker's ID, as the pseudo code in Figure 26 (a) shows. The technique of *Persistent threads* maps multiple jobs to a single worker, but the set of jobs for a worker is still determined by the worker's ID, as shown in Figure 26 (b). Since the placement of workers on SMs is controlled by the hardware schedulers, the job-worker binding makes the placement of jobs on SMs solely depend on the hardware schedulers.

The idea behind SM-centric task selection is to replace the job-worker binding with a binding established between jobs and SMs. As Figure 26 (c) shows, a job queue is built for every SM before the invocation of a kernel function. Inside the kernel function, each worker first figures out on what SM it resides, and then uses the SM ID to fetch the next job in the corresponding job queue to execute. In this way, controlling the placement of a job on a specific SM becomes simple: Just putting that job's ID into the job queue of that SM.

The idea is straightforward. But some complexities must be addressed to implement the idea soundly and efficiently.

Correctness Issues by Hardware Schedulers Through a close look at the pseudocode in Figure 26 (c), one will see that for it to work correctly on a GPU program, the number of workers assigned to an SM must be no fewer than the

[2]In this work, "job" and "task" are interchangable terms, although we tend to use "job" more often when referring to entities in this abstract kernel execution model.

number of jobs assigned to the SM. It is because in that code, one worker on an SM processes only one job assigned to that SM. Some jobs on that SM would be left unprocessed if the number of workers is less than the number of jobs.

However, how many workers are assigned to an SM is determined by the GPU hardware scheduler. Our experiments indicate that the assignment by hardware schedulers is often unpredictable. On a Tesla M2075 with 14 SMs, for instance, when running a kernel with 1400 workers (i.e., CTAs), we observe an uneven distribution of workers: the number of workers per SM varies from 92 to 110. And when running the kernel with 14 workers, some SMs get multiple workers while others get none. Moreover, the worker distribution varies from run to run, displaying lots of randomness.

Such non-determinism jeopardizes the soundness of the basic SM-centric task selection. An option is to allow dynamic job stealing such that workers on one SM can steal jobs left on another SM. It requires more complicated code to be inserted into the GPU kernel to implement the job stealing logic, and hence increases register pressure and reduces parallelism. More importantly, the stealing changes the intended job-to-SM mapping.

4.3.2 Filling-Retreating Scheme

We address the complexity through a *filling-retreating scheme*. This scheme offers a simple way to precisely control the number of active workers on each SM.

The scheme works hand-in-hand with the concept of persistent threads. Similar to persistent threads, with this scheme, a small number of workers are kept alive for each SM throughout the kernel execution. These workers continuously fetch and process the jobs assigned to the SM until the queue gets empty. The tricky part is on how to precisely control the number of active persistent threads (or in our term, active workers) for each SM.

Filling-retreating offers a simple solution. It leverages a common property of GPU schedulers. On GPU, each SM can only support a limited number of active workers at the same time due to hardware limitation. On all GPUs we tested, despite the differences in their schedulers, one common property is that they always try to assign a worker to an SM that can still accommodate some active workers if there is any (rather than putting the worker into a waiting queue of an SM).

Suppose that one SM can support at most m active workers at the same time. In the filling-retreating scheme, a total of $m * M$ workers are created at a kernel launch, where M is the number of SMs in the GPU. Due to the aforementioned common property, each SM gets m workers assigned. This step is the "filling" part of the scheme.

Although the "filling" step ensures every SM gets m workers, as multiple studies have shown [50, 59], having the largest number of workers on an SM is not always the best for maximizing the computing efficiency due to cache and bus contention. This phenomenon is also confirmed in Section 4.6. The "retreating" part of our scheme facilitates flexible adjustment of the number of active workers on an SM. Suppose one wants to have n_{target} active workers per SM. A counter is created for each SM to record the number of workers that have started processing jobs on that SM. Each worker, before starting working on a job, first atomically increases the corresponding counter and then checks whether the counter value already exceeds n_{target}. If so, the worker exits immediately. Figure 27 shows the pseudo code.

The correctness of the filling-retreating scheme relies on the fast distribution of thread blocks by the hardware scheduler. That is, the filling phase should finish before any thread block retreats (i.e., exiting its execution). Otherwise, the hardware scheduler could assign totally more than m workers onto an SM because of the vacancy on that SM formed by the early retreat of some workers on

```
createJobQ4sms(); // on CPU

kernel_smc()
  smID = getSMID();
  workers = workerCounters[smID]++; // atomic
  if (workers > wantedNumPerSM)
    return;
  while (jobID = JobQ[smID].next() != null)
    processJob (jobID);
```

Figure 27: Psuedo code of a GPU kernel in a *filling-retreating* scheme.

it. Correspondingly, some other SMs would get less than m workers assigned. Fortunately, our experiments show that such cases have never happened, plausibly due to the extreme speed of the hardware-based assignment of workers. A check put into the runtime driver could further ensure the condition to hold, the necessity of which is not shown in our experiments.

The benefits of the precise control of the number of active workers on an SM goes beyond helping with the correctness of SM-centric task selection. It also enables a precise control of parallelism on GPU, which facilitates flexible partitions of SMs among co-running kernels shown in Section 4.5.

4.3.3 Implementation

The SM-centric transformation can be easily applied either manually or through a compiler. For proof of concept, we build a prototype source-to-source compiler based on Cetus [55], in which the transformation is implemented as a pass over the input code. The experience taught us the importance of several subtle considerations in the design, which we highlight next before showing the full details of the implementation.

First, the dequeue operation in the while loop in Figure 27 is good for illustrating the basic idea but poor for performance. An atomic operation could cause substantial overhead especially when the work in the loop body is small. When implementing the transformation, it is important to avoid such atomic operations in the while loop. In our design, we circumvent the needs for atomic job fetching

by leveraging a property offered by the filling-retreating scheme: There are precisely N_{target} active workers on an SM. With that property, each active worker only needs to process W/N_{target} jobs, where W is the total number of jobs assigned to the SM. So if we put the job IDs of an SM into an array, the set of jobs for a worker just corresponds to a segment of the array. The starting and ending indices of the segment can be easily attained before the worker enters the job fetching and processing loop. With this improvement, the while loop in Figure 27 can be converted into a simple for loop, iterating the elements in the segment assigned to the worker, and the atomic operation can be hence removed from the loop. In our implementation, we actually use a single array to store the IDs of all jobs. The set of jobs of an SM corresponds to just one section of the array. The position of a job ID in the array hence determines on which SM it will be processed. (Lines 9 to 16 in Listing 4.1 implement this design; explained later.)

Second, the ID of an SM can be obtained efficiently. CUDA, like the C programming language, allows programmers to insert assembly code, which is designed by NVIDIA as an intermediate representation named Parallel Thread Execution (PTX) [75]. It has a special register, %smid, which stores the SM identifier. One "mov" instruction can copy the value in %smid to an integer variable. Line 27 in Listing 4.1 shows the code.

Details In this part, we describe some low-level complexities that our description has skipped. The discussion is based on CUDA, but the implementation can be done for other GPU programming models, such as OpenCL.

On GPU, the spatial scheduling unit is not a thread but a CTA, an array of threads. Correspondingly, the job assignments to processors in our design is in the unit of *job chunks*—the set of jobs executed by a CTA in the original GPU program. In a typical GPU program, the thread ID is used to distinguish jobs, and

one CTA handles one job chunk; the ID of a CTA in the original program is hence treated as the ID of the job chunk that CTA processes.

To minimize changes needed to the original GPU program, we encapsulate most parts of the code for SM-centric transformation into four macros. With them, applying SM-centric transformation involves only several minor changes to the original GPU program. As Listing 4.2 shows, on the CPU-side code, it inserts one macro, *__SMC_init*, before the invocation of a GPU kernel, and appends three arguments to the kernel call. On the GPU-side code, it inserts the calls to two other macros, *__SMC_Begin* and *__SMC_End*, and replaces the appearances of the ID of CTA in the kernel with *__SMC_chunkID*. These can be done easily by the compiler in one pass over the original GPU program.

The above four macros are defined in Listing 4.1. The first, *__SMC_init*, initiates the three variables with the number of workers needed, the array of the desired sequence of IDs of job chunks, and an all-zero counter array to count active workers. The functions used to initiate the first two variables can be provided by the programmer or the optimizing compiler; their definitions depend on the purpose of the specific application of the SM-centric transformation, as Section 4.5 will illustrate. The second macro, *__SMC_Begin*, first calls the fourth macro to get the ID of the SM by reading the particular register, then checks whether the SM already has enough active CTAs. If not, it computes the starting and ending positions of the sets of jobs it should work on, gets into the for loop to process them one by one. The third macro, *__SMC_End*, is trivial, just putting in the ending bracket of the "for" loop in the second macro.

4.3.4 Soundness

At a high level, SM-centric transformation manipulates the association between jobs and processors, and hence alters the mapping between jobs and threads and possibly the execution order of the jobs. As a kind of remapping transforma-

```
1  #define  __SMC_init  \
2  unsigned int * __SMC_workersNeeded = __SMC_numNeeded(); \
3  unsigned int * __SMC_newChunkSeq = __SMC_buildChunkSeq(); \
4  unsigned int * __SMC_workerCount= __SMC_initiateArray();
5
6  #define  __SMC_Begin  \
7  __shared int __SMC_workingCTAs; \
8  __SMC_getSMid; \
9  if(offsetInCTA == 0) \
10   __SMC_workingCTAs = atomicInc (&__SMC_workerCount[__SMC_smid],\
11     INT_MAX); \
12 synchthreads(); \
13 if(__SMCS_workingCTAs >= __SMC_workersNeeded) return; \
14 int __SMC_chunksPerCTA = __SMC_chunksPerSM / __SMC_workersNeeded; \
15 int __SMC_startChunkIDidx = __SMC_smid * __SMC_chunksPerSM + \
16     __SMC_workingCTAs * __SMC_chunksPerCTA;\
17 for (int __SMC_chunkIDidx = __SMC_startChunkIDidx; \
18     __SMC_chunkIDidx< __SMC_startChunkIDidx+__SMC_chunksPerCTA ; \
19     __SMC_chunkIDidx++) { \
20     __SMC_chunkID = __SMC_newChunkSeq[__SMC_chunkIDidx];
21
22 #define  __SMC_End  }
23
24 // get the ID of the current SM
25 #define  __SMC_getSMid  \
26 uint __SMC_smid;\
27 asm("mov.u32 %0, %smid;" : "=r"(__SMC_smid) )
```

Listing 4.1: Macros that materialize SM-centric transformation (N jobs; M SMs).

tion as persistent threads is, for SM-centric transformation to work soundly, the GPU program needs to meet the same conditions as in the case of persistent threads [19, 35]:

(1) The operations by different threads are discriminated only by the thread ID; (2) The execution order of the CTAs does not disturb the correctness of the kernel.

The first condition ensures that SM-centric transformation does not change integrity of a job even though *all* appearances of the CTA ID in a kernel are replaced with the *__SMC_chunkId*. We note that even though current GPU does not migrate CTAs across SMs, the job integrity holds even if CTA migrates—given that the attainment of *__SMC_chunkId* is atomic. The second condition ensures that the new order of execution maintains the meaning of the program.

The two conditions hold for well-formed GPU programs, due to the nature of GPU execution models. At a high level, they are Single-Program-Multiple-Data (SPMD) models; all GPU threads at a kernel launch execute the same function, while their specific operations are determined only by the thread ID. Meanwhile,

```
1   /**** CPU-side code ****/
2   main (){
3     ...
4     __SMC_init;
5     invoke original kernel with three extra arguments:
6           __SMC_chunkCount, __SMC_newChunkSeq, K/M    .
7     ...
8   }
9
10  /**** GPU-side code ****/
11  __global__ kernel (...,
12  unsigned int *__SMC_chunkCount, unsigned int *__SMC_newChunkSeq,
13  unsigned int __SMC_chunksPerSM)
14  {
15    __SMC_Begin
16    // the original kernel with the ID of CTA replaced with __SMC_chunkID
17    ... ...
18    __SMC_End
19  }
```

Listing 4.2: GPU program after SM-centric transformation (N vector elements; K job chunks; M SMs).

for a GPU program to work properly, it should not rely on the execution order of CTAs, because due to the non-determinism in CTA scheduling on GPU, it is hard to know what order would be taken in a run. Free from data race helps ensure the conditions hold. Recent years have seen a number of studies on data race detection for GPU [13, 101], which could serve as part of the automatic check of the applicability of SM-centric transformation.

4.4 Uses and Complexities

By enabling program-level spatial scheduling, SM-centric transformation opens up some new opportunities for GPU optimizations. This section discusses some of them, and examines the main complexities associated with these new opportunities.

4.4.1 Example Uses

SM Partition for Multi-kernel Co-runs It has been observed that many GPU kernels exhibit sublinear speedups when the number of SMs used for the kernel increases [4, 72]. As a result, simulations have shown that if the set of SMs in a GPU can be partitioned such that different subsets of SMs work for different

kernels concurrently, the system often gives higher throughput and the kernels manifest better overall responsiveness [4]. However, such partitions have not been feasible in practice for lack of scheduler controllability. On NVIDIA GPUs, for instance, when two kernels are launched concurrently (each usually has many CTAs), their CTAs are assigned to all SMs. And if the threads by one kernel already use too much register or shared memory on an SM, before its completion, the other kernel cannot start, hence resulting in a serial execution of the two kernels.

With minor extension to the SM-centric transformation, partitions of SMs among concurrent kernels becomes possible. For instance, if we want the first 6 SMs to work for kernel f_1, and the remaining 8 SMs for kernel f_2, we can set the mapping array used in SM-centric transformation such that all jobs of f_1 map to the first 6 SMs and those of f_2 to the other 8 SMs. When the two kernels get launched, the GPU scheduler still assigns CTAs of both kernels to every SM. However, a statement is inserted in each kernel after obtaining the SM ID, which checks whether the ID of this SM is one of the SMs supposed to work for this kernel. If not, the CTA returns immediately so that the SM can work for the other kernel.

Affinity-Based Scheduling Many GPU applications have inherent non-uniform data interactions, such as the MD example mentioned in Section 4.2. It causes non-uniform data sharing among job chunks. Following the concepts on traditional CPU [103], we state that two job chunks have good reference affinity if they share lots of data. As Section 4.2 mentions, each SM has an on-chip cache. So, if we can manage to assign onto the same SM the job chunks with good affinity, we may enhance the performance of the cache. SM-centric transformation makes this affinity-based scheduling possible.

4.4.2 Complexities

There are some complexities for applying SM-centric transformation to the use cases mentioned earlier.

For SM partition for co-runs, the key is to decide the best partition. For affinity-based scheduling, the key is to compute the affinity among job chunks and then group job chunks accordingly.

Additionally, there is a common complexity existed in both use cases: determining the suitable number of active CTAs for a kernel. As some studies have shown [50], creating the maximum number of CTAs that an SM can hold often gives suboptimal performance, because of cache and bus contention among them. The CTA aggregation employed in SM-centric transformation allows flexible control of the number of active CTAs for a kernel.

However, determining the suitable numbers of active CTAs is challenging. It depends on many factors, including the interaction between SM partitioning and data locality, program inputs, kernels' resource requirement and so on. Moreover, when coupling with the various methods to partition SMs (for co-runs), they could result in a large search space. For 2 kernels on a 14-SM GPU, if an SM can support at most 6 active CTAs, the search space contains $6 \times 6 \times 14 = 504$ cases.

4.5 Designs for Validation

In this section, we describe our design to address the complexities listed in the previous section. Our goal is to validate the practical value of the spatial scheduling enabled by SM-centric transformation, rather than to find the best solution to those complexities. Simplicity and practicality are the principles in our design. The rationale is that if the enabled spatial scheduling could bring substantial benefits with minimum support, the promise of the technique is validated.

4.5.1 Optimal Configuration Search

We first discuss the challenges for determining the best number of active CTAs (i.e., the n_{target} mentioned in Section 4.3.2, which is also called parallelism control) for a kernel and for finding the best partition of SMs between co-running kernels. We call these parameters together as a configuration in our discussion. The difficulty is that the space of the configuration values is large and the best configuration depends on many factors. It is often too costly to try every possible configuration at runtime. We employ the standard sampling method to efficiently approximate the best configuration. When a kernel is inside a loop, the sampling may happen during the first several iterations; otherwise, the sampling may happen offline or across runs.

Because SM partition mainly affects interactions across SMs, while the parallelism control is mainly related with resource usage inside an SM, we observe that the optimal level of parallelism for a kernel is only loosely connected with how we partition SMs. Hence our search scheme first evenly partitions the SMs among kernels, and tries to find the appropriate numbers of active CTAs for each kernel. It starts with the maximum CTAs supported by an SM for the kernel (no larger than 8), and decreases the number by 1 in each iteration until it observes decreased performance or the number reaches 1. This is a typical process of hill climbing. After that, our search scheme fixes the CTA numbers but adjusts SM partition by setting the number of SMs assigned to a co-run kernel from 1 to the maximum-1 (in a step size of three) while the rest SMs are used for the other co-running kernel. As a prior study [72] does, this work considers only co-runs of two kernels. Based on the sampled data, the optimal partition is approximated through interpolation.

Like other online sampling-based approaches, our search scheme cannot work well when different iterations behave dramatically differently. Combining the

sampling approach with domain knowledge about the behavior patterns of the program may help, which is out of the scope of this work. In our experiment, we encountered only one such program, *reduction*. We did not give special treatment to it. The results in Section 4.6 show that even though the sampling method finds only suboptimal configurations for some programs, the overall benefits are still substantial, confirming the value of the SM-centric transformation.

4.5.2 Affinity-Based Scheduling

To implement the affinity-based scheduling, we model the scheduling problem as a graph partitioning problem. The modeling consists of two steps: graph construction and graph partitioning.

Graph Construction This step establishes a set of graphs named *affinity graphs*, in which, each vertex represents a job chunk and each edge weight represents the affinity score between two job chunks. Affinity score is defined as follows. Let S_1 and S_2 be the set of data blocks assessed by two job chunks J_1 and J_2 respectively. Their affinity score is $\frac{|S1 \cap S2|}{|S1 \cup S2|}$. There is no edge between two vertices when their affinity score is less than a threshold (0.05 in our experiment). If the affinity score is too small, there is only a small amount of data sharing and its effects on performance is negligible; ignoring them often breaks one affinity graph into multiple smaller graphs, allowing efficient graph partitioning in the next step.

Compiler techniques exist for analyzing working sets [81] for regular applications. On irregular applications, it is challenging as data access patterns may be unknown until run time. Runtime inspection techniques have been proposed to analyze data access patterns [65]. For GPU programs, prior work has shown the feasibility to employ CPU to implement the inspection asynchronously when GPU is executing the kernel [99]. In this work, we used simple synchronous par-

allel inspection for irregular programs. But the asynchronous method may further reduce the overhead.

Graph Partitioning Given M SMs, this step partitions the set of vertices into M equal-size clusters. The job chunks corresponding to a cluster are scheduled to one single SM. This problem is known to be NP-hard. There are some existing heuristic algorithms, but we find them costly. Instead, we design a random and lightweight algorithm. Its basic idea is to select a seed vertex for each cluster and greedily enlarge each cluster to include the vertices that have high affinity scores with the selected vertices. The algorithm has three steps. (1) *Seeds selection.* Selecting the seed vertices is important for the partitioning quality; we try to minimize the affinity among them. The initial seed set is formed by randomly selecting a vertex from each of the affinity graphs. If the number of affinity graphs is no less than M, only M of them are randomly selected. If there are less than M seeds in the set, we iterate over the remaining vertices until we find one, whose affinity scores with all current seeds are smaller than a threshold (initialized to 0) and add it to the seed set. This step stops once we get M seeds. After iterating all vertices if we still need more seeds, we increase the threshold by 0.1 and start the next round of search. Ten rounds are needed at most as the threshold would grow to 1, the largest possible affinity score. In practice, we have not seen the need for more than 1 round. After the seeds selection, we have M clusters, each containing 1 vertex. (2) *Sorted lists construction.* For each seed vertex T_i, we create a descending list of all the vertices that fall into the same affinity graph as T_i. (3) *Cluster enlargement.* This step repetitively iterates through all clusters until all vertices are partitioned. In each iteration, it randomly selects a vertex from the current cluster, and includes the vertex that, among all remaining vertices, has the largest affinity score with this vertex, which can be determined in constant time with the sorted lists produced in step 2.

In our implementation, this graph partitioning happens in parallel on CPU. Its time complexity, in the worst case when all vertices fall into one graph, is $O(N^2 logN)$ (N for number of job chunks). But in practice, as graphs are never very large, the algorithm terminates quickly shown in the next section.

4.6 Evaluations

We focus our experiments on answering the following main questions:

- How much potential does spatial scheduling enabled by SM-centric transformation have?
- How much overhead does SM-centric transformation have?
- How much benefit can it bring in practice with the simple support outlined in the previous section?

To that end, we implement the two use cases of STM-centric transformation as described in Section 4.4: One is SM partition for multi-kernel co-runs, and the other is affinity-based scheduling for single-kernel runs. The implementation integrates the solutions described in Section 4.5. For comparison, we also implement the persistent threads with the best efforts to support these two use cases.

4.6.1 Methodology

Benchmarks Given that the focus of our use cases are on enhancing memory performance, we need a set of memory intensive programs for the validation. Meanwhile, for a comprehensive assessment of the applicability of our techniques, the benchmark set should consist of programs of a broad range of domains, and have a good coverage of both regular and irregular programs. For these reasons, we select nine benchmarks to form our test set. As Table 4 shows,

Table 4: Benchmarks

Benchmark	Source	Description	Irregular
irreg	Maryland [38]	partial diff. solver	Y
nbf	Maryland [38]	force field	Y
md	SHOC [23]	molecular dynamics	Y
spmv	SHOC [23]	sparse matrix vector multi.	Y
cfd	Rodinia [17]	finite volume solver	Y
nn	Rodinia [17]	nearest neighbor	N
pf	Rodinia [17]	dynamic programming	N
mm	CUDA SDK [69]	dense matrix multiplication	N
reduce	CUDASDK [69]	reduction	N

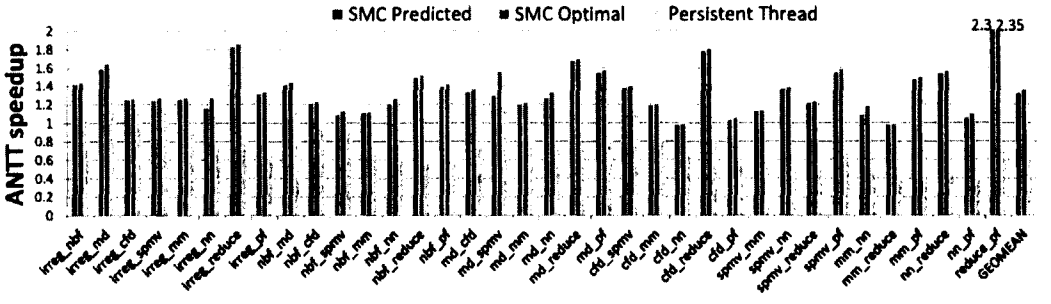


Figure 28: Speedup of average normalized turnaround time.

these programs come from four benchmark suites, cover a broad set of domains, and include a similar number of regular and irregular programs. Those irregular benchmarks impose special challenges for GPGPU optimization, and have drawn a lot of attention from the community recently [14, 57, 67, 68, 93, 99].

We give a brief description for these benchmarks. IRREG (a partial differential solver kernel) and NBF (a molecular dynamics kernel) were rewritten to CUDA from C benchmarks [38]. These two benchmarks were studied heavily by previous work [27, 36, 85, 92]. MD and SPMV are both from the SHOC benchmark suite developed by Oak Ridge National Laboratory [23]. CFD from Rodinia benchmark suite [17] simulates fluid dynamics. MM and REDUCE taken from the CUDA SDK samples represent two compute-intensive applications used widely in real-world. We also take NN and PF from the popular Rodinia benchmark suite for a broader coverage.

Co-runs of Kernels As current GPUs cannot support the co-existence of two different contexts yet, following prior work [72], we combine two programs into one and use two separate CUDA streams to execute the kernels of the two original programs. Since which kernels run together depends on the practical context, we co-run each pair of the benchmarks for a comprehensive coverage. We use two metrics, System Throughput (STP) and Average Normalized Turnaround Time (ANTT), proposed in [29] and used in [72]. STP shows overall throughput of the whole system, and ANTT shows programs' responsiveness. We measure the execution time of kernel executions and the extra overhead introduced by the transformation (if any) for the calculation of STP and ANTT. Since we are only interested in the overlapped execution, we modify the approach proposed by Tuck and Tullsen [88] and immediately invoke a kernel after it finishes until both kernels are invoked at least 7 times. The last instance of the kernel invocation that finishes later than the other co-run kernel is discarded, because the execution of the last instance of the kernel invocation is not fully overlapped.

Versions based on Persistent Threads We compare SM-centric transformation with persistent threads using both SM partition and affinity-based scheduling. As aforementioned, persistent threads by itself cannot directly dictate mappings between jobs and SMs. But with careful designs, it could still support SM partition and affinity-based scheduling, although the support is very limited and requires an awkward implementation. Our specific implementations are as follows.

For SM partition between two co-running kernels, we generate $N_1 (1 \leq N_1 \leq M)$ persistent CTAs for kernel 1, and $(M - N_1)$ persistent CTAs for kernel 2 (where M is the number of SMs in the GPU). In this way, if the hardware scheduler happens to assign one CTA onto each SM, the two kernels would run on different sets of SMs, and the SM partition is materialized. Given that an even distribution is not guaranteed by hardware schedulers, in our experiments, we repeat the experi-

ments many times and use only the results when the distribution happens to be even (performance under uneven distributions is much worse as some SMs are left idle). In order to maintain a good amount of parallelism, each CTA is set to the largest allowable size for the particular kernel, as prior usage of persistent threads often does [19,95]. Our experiments enumerate all possible partitions (i.e., all values of N_1), and the best performance in these settings is used to compare with the performance of SM-centric transformation results.

We employ a similar idea to let persistent threads support affinity-based scheduling of single-kernel runs. The launch of a kernel creates M (i.e., number of SMs) persistent CTAs. We again use the performance measured only in the runs where the CTAs are evenly distributed on the SMs. When creating the job queue for a persistent CTA, we try to put into the queue the jobs from the same affinity group as identified with the method in Section 4.5.2. We again make each persistent CTA as large as allowed such that the maximum number of jobs from the CTA queue could get concurrently executed by the CTA. This method, in effect, makes the jobs run concurrently on the same SM, just as what affinity-based scheduling aims to achieve—but only to a limited degree, subject to the number of jobs a CTA can concurrently execute.

Machine Environment We run all workloads on an NVIDIA M2075 GPU with CUDA runtime 4.2, compiled by NVCC with the highest optimization level. The host machine has an Intel 8-core Xeon X5672 CPU and 48 GB main memory and runs 64-bit Redhat enterprise 6.2. Without notice, each reported timing result is an average of 10 repeated measurements, and includes all overhead incurred by the transformed code.

4.6.2 Results in Co-Runs

Figure 28 shows the speedup in terms of ANTT brought by the optimized co-runs respectively supported with SM-centric transformation and persistent threads. The baseline is the traditional and default way to concurrently execute the original kernels. The speedup is defined as the ratio of the optimized ANTT to the original ANTT. "SMC Predicted" and "SMC Optimal" represent the speedups from the SM-centric transformation with, respectively, the parameters predicted by the online model and the best parameters found through offline exhaustive search. We observe the potential speedup because of SM-centric transformation 1.36X. Our prediction model successfully exploits most of the potential by providing 1.33X speedup on average. Three co-runs benefit from the SM-centric transformation substantially with a potential of more than 1.8X speedup. The results validate that SM-centric transformation with the prediction model better exploits the SM and cache resources than the default co-runs. We also observe that the improvement of ANTTs varies across benchmarks. In some cases (e.g., the co-run of mm and reduce), the optimized co-runs have around 2% slowdown. There are two plausible reasons. First, the kernels already have good scalability and hence reducing the number of SMs allocated to them significantly degrades their performance. Second, since those kernels efficiently use shared memory, they do not heavily rely on L1 cache's performance. For example, when MM's thread blocks size is 256, one matrix element is reused 16 times after being loaded into shared memory.

Persistent threads perform much worse than SM-centric transformation. Its best partition leads to more than 50% ANTT degradation for 3 co-run programs. On average, we observe 17% slowdown. The main reason comes from the rigid control of parallelism in persistent threads. As the previous subsection describes, without the capability for a direct control of the job-to-SM mapping, the design of

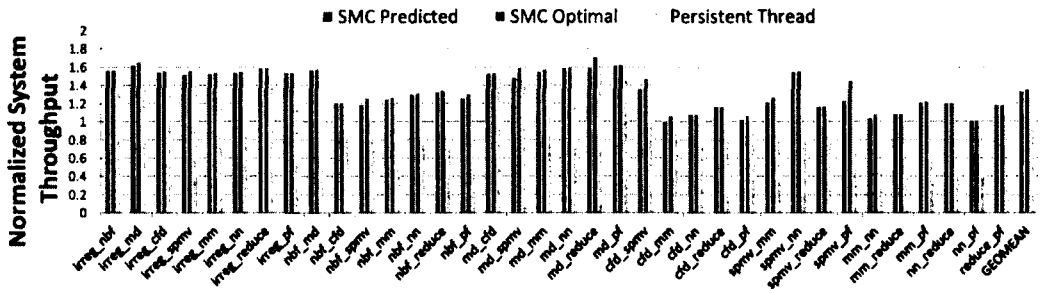


Figure 29: Improvement on system throughput.

persistent threads support is subject to some restrictions on the number of a CTAs and their size, which cause suboptimal performance on the kernels.

Figure 29 provides the results on system throughput. The baseline is of the same case in Figure 28. SM-centric transformation offers up to 71% (an average of 37%) improvement on STP. The predicted configurations exploit the potential well by providing an average improvement of 33%. We did not observe any throughput degradation for the optimized co-runs, but some co-runs (e.g., cfd and mm) have trivial throughput improvement due to the same reasons as explained for the worsened ANTTs. As known [29], ANTT and STP measure different aspects of a co-run execution; a better ANTT does not always mean a better STP. For instance, the co-run of reduce and pf has the largest ANTT speedup of 2.3X, but its STP improvement is below the average.

Different from the results on ANTT, persistent threads produce an average of 11% STP improvement. The influence of persistent threads on ANTT varies greatly across co-run programs, yielding results between 63% slowdown and 64% improvement. As explained for the increased ANTT, We observe non-trivial throughput loss for some programs because of the suboptimal level of parallelism. Overall, the results showed that SM-centric transformation is a much better choice for SM partitioning than persistent threads.

The SM-centric results also indicate that the simple method for predicting configurations outlined in Section 4.5.1 is sufficient for SM-centric transformation to effectively support SM partition. To get a direct measure of the method's

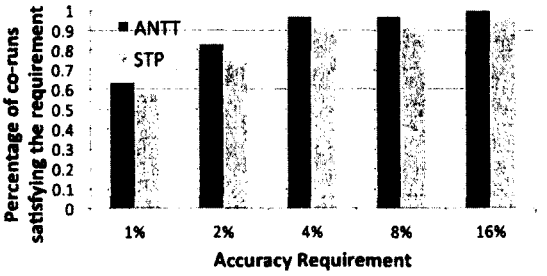


Figure 30: Prediction accuracy

effectiveness, we report in Figure 30 its accuracy in predicting the suitable configurations. The percentage on the X axis shows the accuracy requirement of the predicted configuration. To be more specific, $P\%$ means that the predicted configuration outperforms at least $N \times (1 - P\%)$ configurations, where $N (N = 36$ in this evaluation) is the total number of configurations. The bar height shows the percentage of co-runs whose predicted configuration satisfies the accuracy requirement. So the bars on the right should be higher than the bars on the left, because a larger percentage on the X axis indicates a more relaxed requirement. For ANTT, when the accuracy requirement is 1%, 63% of co-runs satisfy it. Note that 1% is a harsh requirement, as only the optimal configuration can satisfy it in a limited configuration space. If we relax the requirement to 2%, 83% of co-runs satisfy it, showing a high prediction accuracy. When the requirement is 16%, we notice that the predicted configuration of every co-run satisfies the requirement. For STP, the prediction accuracy is a bit lower, but over 90% of co-runs satisfy the accuracy requirement of 4%. The results echo our improvement on ANTT and STP and show that a simple online model suffices to yield reasonably good configurations.

4.6.3 Results in Single-Kernel Runs

We also evaluate persistent threads and SM-centric transformations on single-kernel runs. We consider 4 programs (md, irreg, cfd and nbf), as they show a

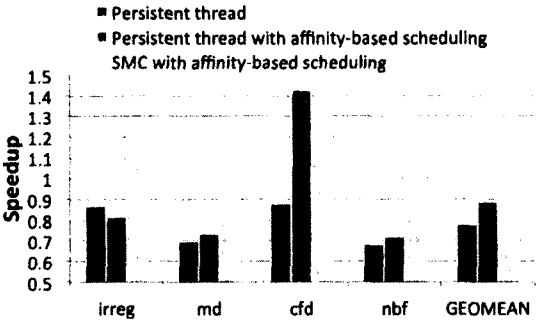


Figure 31: Speedups of single-kernel runs.

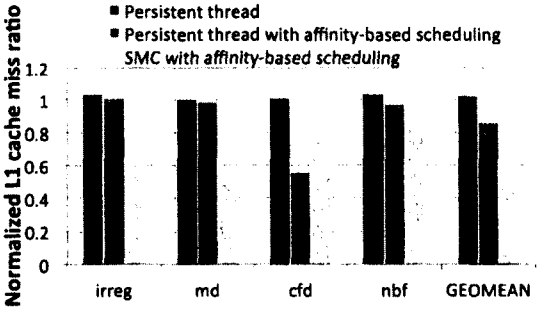


Figure 32: Normalized L1 miss ratios.

significant level of non-uniform data sharing and rely heavily on the data cache's performance due to their irregular memory access pattern.

Figure 31 provides the speedup results for single-kernel runs of four benchmarks over the original code. Without affinity-based scheduling, persistent threads suffer from insufficient parallelism and produce 22% performance degradation. Affinity-based scheduling improves its performance and reduces the average degradation to 15%. The results indicate that persistent threads, unlike SM-centric transformation, fails to achieve a good balance between parallelism and locality: Keeping one active thread block on each SM enables scheduling jobs with lots of data sharing to one SM, but due to the limitation of the block size, does not have enough concurrent active threads to fully explore the computing power. On the contrary, SM-centric transformation's precise control enables us to find a better trade-off between parallelism and locality, leading to an average of 21% speedup.

Figure 32 shows the L1 cache performance improvement obtained through CUDA hardware performance monitors. The reduction of the cache miss ratios

shows the trends largely aligning with the speedup trends. It confirms that L1 cache performance is critical to irregular applications, and the parallelism control and affinity-based scheduling enabled by SM-centric transformation exploit L1 data cache more effectively than the default scheduling does. CFD is an exception, on which, the SM-centric approach performs less well than the persistent threads with affinity-based scheduling. A plausible reason is the effects of warp scheduling, which is out of the control of SM-centric scheduling but could sometimes affect the cache performance substantially.

4.6.4 Overhead from the SM-centric Transformation

SM-centric transformation adds extra code to the kernels. To quantify the overhead, for each benchmark we run the transformed kernel (with the same number of active threads as the default runs of the original kernels have) but without affinity-based scheduling, whose execution time is denoted as T_{trans}. The overhead is defined as $(T_{trans} - T_{org})/T_{org}$, where T_{org} is the execution time of the original kernel. Figure 33 provides the overhead results. We notice that the overhead can be non-trivial for some benchmarks (e.g., 6.5% for pf) due to two reasons. First, the transformation introduces atomic operations and extra memory accesses to obtain the mapping decision data. Second, the aggregation (i.e., the enhanced version of the transformation) introduces a loop, which does not exist in the original kernels. On average, the overhead from the transformation is 2.8%, but as the previous results show, the overhead is substantially outweighed by the overall benefits.

4.7 Discussion

Currently, the SM-centric optimization works only on CUDA programs. The reason is that other GPU programming models, such as OpenCL, do not support run-

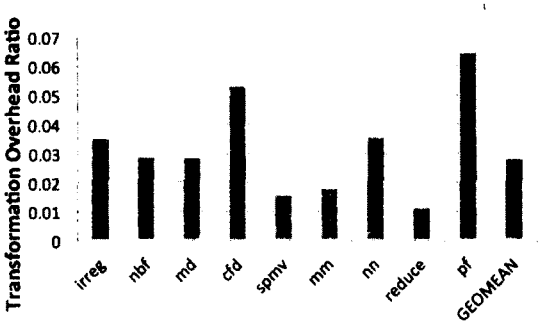


Figure 33: The percentage of overhead from SM-centric transformation.

time retrieval of compute unit identifier yet. We hope that this research provides enough evidence for the need of such a functionality. Once OpenCL provides a similar interface, the proposed optimization techniques can be easily extended to cover OpenCL programs.

OpenCL, starting from specification 1.2, introduced the concept of sub-device, which wraps a subset of the computing unites. While the idea behind it is also to enable resource partition among different applications, it is not yet supported by main-stream GPU vendors. Our SM-centric optimization, as an alternative approach for the partitioning, not only works for current NVIDIA GPU cards, but also enables programmers to control which SMs should be in the same sub-device.

The job selection component determines the job-to-SM mapping before the kernel is invoked. This fixed mapping could incur load-unbalance due to the non-uniformity in SM processing capability and jobs. It is possible to detect the load-unbalance during the sampling phase. The framework can then invoke the original kernel if load-unbalance happens. Some kernels may change dramatically across invocations in terms of execution time and memory access intensity, rendering great challenges to any sampling based approach. Combination with program phase detection and prediction [82, 83] could help address such complexities.

4.8 Related Work

Prior software methods on circumventing the hardware restrictions for task scheduling on GPU mainly concentrate on persistent threads with either static or dynamic (e.g., job stealing) partition of task sets [6, 19, 35, 89, 95], which has been covered in our previous sections.

There are some studies on changing hardware schedulers for performance, such as the large warp architecture by Narasiman and others [66], the two-level warp scheduler (and interactions with prefetching) by Jog and his colleagues [46, 47], and the thread block scheduler by Kayiran and others [50]. The SM-centric scheduling is a software solution to the restrictions of hardware schedulers, orthogonal to these hardware approaches. There are some software scheduling works published before, the focus of which have been dealing with the load balance between CPUs and GPUs through task scheduling [11, 60].

Recent years have seen an increasing interest in supporting concurrent executions of GPU kernels. Pai et al. [72] observed significant resource underutilization during concurrent kernel executions. They proposed elastic kernels that have fine-grained controls over their resource usage to balance resource usage among concurrent kernels. According to the authors of elastic kernels, the technique does not control SM partitions, and cannot be applied to kernels that use shared memory [72]. The spatial scheduling enabled in this work is complementary to elastic kernels, in the sense that it is not subject to the shared-memory limitation, and it improves co-run performance from a different angle, spatial scheduling. These two techniques can be used together. Adriaens and others [4] proposed hardware extensions to partition SMs to different applications for more efficient resource utilization, and evaluated it on a simulator. Zhong and He [102] proposed a runtime system, named Kernelet, which slices kernels into sub-kernels and schedule them for better resource control. It does not enable

spatial scheduling of GPU. CTA aggregation itself is not new. Earlier work has used a similar idea to control resources a kernel uses [72]. It is for the first time used for supporting spatial scheduling.

4.9 Summary

This chapter presents SM-centric transformation, a simple method that for the first time offers a systematic solution to enable program-level spatial scheduling on GPU. It reveals the potential of the enabled scheduling control for executions of both single-kernel runs and multi-kernel co-runs. It lists some main challenges for leveraging spatial scheduling on GPU, and develops a set of practical solutions. It opens up opportunities for leveraging scheduling for optimizing GPU executions.

Chapter 5

Limitations and Future Work

We briefly discuss some limitations and possible future work to address them.

5.1 Application Coverage

All the applications considered in this work are iterative: The kernel functions are invoked many times until some condition is met. As such, although the overhead from the data layout usually outweighs the benefit for one kernel invocation, it can be amortized by the many invocations, as shown by the speedup results. However, some applications (e.g.,breadth-first search) may have few iterations, depending on the inputs, and hence may not benefit from the proposed approaches. One promising direction is to quantify the benefit and overhead from the optimizations. Given an application, we can estimate the tradeoff between the benefit and overhead of an optimization and decide whether or not to apply it.

There are two challenges if we pursue in this direction. First, we need to build a reasonably accurate performance model to estimate the overhead of data layout change and the application's performance given a specific layout. Performance modeling, however, is shown to be a hard problem for complicated heterogeneous systems. Some state-of-the-art performance models yield estimation errors up to 50%, which are comparable to the benefit from some memory optimizations.

Such accuracies make them hard to use in our context. Furthermore, current performance models typically ignore data caches and treat non-coalesced memory accesses naively for simplification, thereby not suitable for irregular applications.

Second, we need to make the performance estimation fast. Current performance models only consider simple input features, such as data size, array dimension and so on. For our purpose, the performance model should take into account the delicate memory behavior changes due to the layout differences. Modeling such behaviors is non-trivial and may itself take too much time to apply online.

5.2 Automatic Optimization

Most of our proposed optimizations are implemented in libraries. To optimize the programs, the programmers need to pinpoint the statements that cause irregular memory accesses and insert corresponding library calls. Despite our efforts to make the library easy to use, the process unavoidably increases the burden of programmers.

It is possible to automate the optimizations through two steps. The first step locates the points in the program that need to be optimized. Since we focus on indirect memory accesses, the compiler can find out all the occurrences of the pattern $A[P[id]]$. But the appearance of such a pattern does not necessarily mean the need for optimization, as the values in P may indicate regular (or slightly irregular) accesses. To refine the selection, we may use offline profiling to find the statements that cause great irregularity.

The second step automatically transforms the target statements to activate layout reorganization and use the optimized layout. We see no challenges in this part; a simple pass during compilation can do the job.

5.3 Energy Concerns

Some of the proposed optimizations work on a pipelined engine, which uses one processor to reorganize the data to help another processor. This scheme raises concerns on energy cost, due to the introduction of an additional processor as the helper. Since the goal of the dissertation is to minimize overall execution time, we did not measure energy consumption. But we contend that the pipelined engine may not necessarily increase energy consumption, because the optimized layout reduces the total execution time and off-chip memory accesses, thereby lowering the energy consumption on the execution processor.

In the future, we can change the optimization goal to the minimization of energy cost. We will study the potential by measuring the ratio of energy consumption from irregular memory accesses to that of the whole application. We will quantify the energy consumption characteristics of different optimizations, design an online method to select the appropriate optimization and tune relevant parameters, such as the set of arrays to optimize and their corresponding optimization ratios.

Chapter 6

Conclusion

Heterogeneous systems, in which the CPU runs sequential workloads and the GPU runs parallel workloads, already become mainstream. While such systems show tremendous throughput improvement over homogeneous systems for regular applications, efficiently handling irregular applications is still an open problem. One serious problem is the irregular memory accesses commonly seen in many applications, whose pattern can only be determined during runtime. Since the hardware fail to coalesce the memory accesses or reuse data in caches, ineffective off-chip memory bandwidth drags down the overall system performance.

This work systematically explores matching the non-uniformity of software with that of hardware to tap into the full potential of heterogeneous systems. To address non-coalesced memory accesses on GPU, in Chapter 2, we analyzed the complexity of minimizing non-coalesced memory accesses without extra space overhead. We proposed several optimization algorithms to make a good trade-off among time, space cost and complexity. We designed an online algorithm selector to adapt to inputs and hardware settings. In chapter 3, we proposed asynchronous data transformation to address the dilemma between transformation overhead and benefit. By decomposing the transformation into analysis and repositioning, we circumvent the critical data dependences and offload the heavy analysis component from the critical path. In Chapter 4, we provide scheduling

support to enable complete control of the mapping between jobs and processors. We leveraged the software scheduling to optimize non-uniform co-run workloads and non-uniform data sharing within one application.

This dissertation has introduced new perspectives for memory optimization on many-core heterogeneous systems. It is the first of its kind to systematically consider space overhead for data reorganization. The proposed asynchronous transformation framework and the software-level scheduling support open up many opportunities to new optimizations.

Bibliography

[1] libpfm4. `http://perfmon2.sourceforge.net/docs.html`.

[2] NVIDIA CUDA. `http://www.nvidia.com/cuda`.

[3] NVIDIA CUDA. `http://www.nvidia.com/cuda`.

[4] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The Case for GPGPU Spatial Multitasking. In *International Symposium on High Performance Computer Architecture*, 2012.

[5] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, August 2006.

[6] T. Aila and S. Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, 2009.

[7] J. A. Anderson, C. D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *J. Comput. Phys.*, 227(10):5342–5359, May 2008.

[8] P. Bakkum and K. Skadron. Accelerating sql database operations on a gpu with cuda. In *Proceedings of the 3rd Workshop on General-Purpose*

Computation on Graphics Processing Units, GPGPU '10, pages 94–103, New York, NY, USA, 2010. ACM.

[9] 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'08: Proceedings of the 22nd Annual International Conference on Supercomputing*, pages 225–234, 2008.

[10] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11, New York, NY, USA, 2009. ACM.

[11] M. E. Belviranli, L. N. Bhuyan, and R. Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Trans. Archit. Code Optim.*, 9(4), Jan. 2013.

[12] M. Berger and S. Bokhari. A partitioning strategy for non-uniform problems on multiprocessors. *IEEE Trans. Computers*, 37(12):570–580, 1987.

[13] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson. Gpuverify: A verifier for gpu kernels. In *OOPSLA*, 2012.

[14] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on gpus. In *IISWC*, pages 141–151, 2012.

[15] S. Carrillo, J. Siegel, and X. Li. A control-structure splitting optimization for gpgpu. In *Proceedings of ACM Computing Frontiers*, 2009.

[16] G. C. Cascaval. *Compile-time Performance Prediction of Scientific Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 2000.

[17] 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.

[18] S. Che, J. W. Sheaffer, and K. Skadron. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *Proceedings of the ACM/IEEE conference on Supercomputing*, 2011.

[19] L. Chen, O. Villa, S. Krishnamoorthy, and G. Gao. Dynamic load balancing on single-and multi-gpu systems. In *IPDPS*, 2010.

[20] T. M. Chilimbi and R. Shaham. Cache-conscious coallocation of hot data streams. In *Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation*, 2006.

[21] A. Corrigan, F. Camelli, R. Lohner, and J. Wallin. Running unstructured grid based cfd solvers on modern graphics hardware. In *Proceedings of the 19th AIAA Computational Fluid Dynamics*, 2009.

[22] A. Danalis, G. Marin, C. McCurdy, J. Meredith, P. Roth, K. Spafford, V. Tipparaju, and J. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. 2010.

[23] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *GPGPU*, 2010.

[24] R. Das, D. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured euler solver using software primitives. In *Proceedings of the 30th Aerospace Science Meeting*, Reno, Navada, January 1992.

[25] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *J. Parallel Distrib. Comput.*, 22(3):462–478, Sept. 1994.

[26] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *PLDI*, pages 229–241, 1999.

[27] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *PLDI*, 1999.

[28] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *Journal of Parallel and Distributed Computing*, 64(1):108–134, 2004.

[29] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3), May 2008.

[30] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. *Proc. VLDB Endow.*, 3(1-2):670–680, Sept. 2010.

[31] W. Fung, I. Sham, G. Yuan, and T. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society.

[32] A. Geist. Paving the roadmap to exascale. *SciDAC Review*, 2010.

[33] Z. Guo and X. Shen. Fine-grained treatment to synchronizations in gpu-to-cpu translation. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, 2011.

[34] Z. Guo, E. Zhang, and X. Shen. Correctly treating synchronizations in compiling fine-grained spmd-threaded programs for cpu. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 2011.

[35] K. Gupta, J. A. Stuart, and J. D. Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *Innovative Parallel Computing*, 2012.

[36] H. Han and C. W. Tseng. Improving locality for adaptive irregular scientific codes. In *LCPC*, 2000.

[37] H. Han and C.-W. Tseng. Exploiting locality for irregular scientific codes. *IEEE Transactions on Parallel Distributed Systems*, 17(7):606–618, 2006.

[38] H. Han and C.-W. Tseng. Exploiting locality for irregular scientific codes. *TPDS*, 17(7):606–618, 2006.

[39] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning*. Springer, 2001.

[40] S. Herbert and D. Marculescu. Characterizing chip-multiprocessor variability-tolerance. In *DAC*, 2008.

[41] A. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: portable stream programming on graphics engines. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[42] S. Huang, S. Xiao, and W. Feng. On the energy efficiency of graphics processing units for scientific computing. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–8, 2009.

[43] X. Huo, V. Ravi, W. Ma, and G. Agrawal. An execution strategy and optimized runtime support for parallelizing irregular reductions on modern gpus. In *ICS*, 2011.

[44] W. Jia, K. A. Shaw, and M. Martonosi. Characterizing and improving the use of demand-fetched caches in gpus. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, 2012.

[45] Y. Jo and M. KulKarni. Enhancing locality for recursive traversals of recursive structures. In *OOPSLA*, 2011.

[46] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Orchestrated scheduling and prefetching for gpgpus. In *ISCA*, 2013.

[47] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU performance. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[48] M. Kandemir. A compiler technique for improving whole-program locality. In *POPL*, 2001.

[49] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. In *Proceedings of International Conference on Parallel Processing*, August 1995.

[50] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *PACT*, 2013.

[51] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. Opencl as a unified programming model for heterogeneous cpu/gpu clusters. In *PPoPP*, 2012.

[52] S. Kim, H. Han, and K. Choe. Region-based parallelization of irregular reductions on explicitly managed memory hierarchies. *Journal of Supercomputing*, 2009.

[53] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic parallelism benefits from data partitioning. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 233–243, 2008.

[54] J. Lee, V. Sathisha, M. J. Schulte, K. Compton, and N. S. Kim. Improving throughput of power-constrained gpus using dynamic voltage/frequency and core scaling. In *PACT*, 2011.

[55] S. Lee, T. Johnson, and R. Eigenmann. Cetus - an extensible compiler infrastructure for source-to-source transformation. In *In Proceedings of the 16th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 539–553, 2003.

[56] S. Lee, S. Min, and R. Eigenmann. Openmp to gpgpu: A compiler framework for automatic translation and optimization. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2009.

[57] J. Li, G. Tan, M. Chen, and N. Sun. Smat: an input adaptive auto-tuner for sparse matrix-vector multiplication. In *PLDI*, 2013.

[58] W. Liu and A. Sherman. Comparative analysis of the cuthill-mckee and the reverse cuthill-mckee ordering algorithms for sparse matrices. *SIAM J. Numerical Analysis*, 13(2), April 1976.

[59] 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.

[60] C. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO*, 2009.

[61] G. Marin, G. Jin, and J. Mellor-Crummey. Managing locality in grand challenge applications: a case study of the gyrokinetic toroidal code. 2008.

[62] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications. In *Proceedings of International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[63] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *ISCA*, 2010.

[64] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. *SIGPLAN Not.*, 47(8):117–128, Feb. 2012.

[65] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *PACT*, 1999.

[66] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving gpu performance via large warps and two-level warp scheduling. In *MICRO*, 2011.

[67] R. Nasre, M. Burtscher, and K. Pingali. Data-driven versus topology-driven irregular computations on gpus. In *IPDPS*, pages 463–474, 2013.

[68] R. Nasre, M. Burtscher, and K. Pingali. Morph algorithms on gpus. In *PPOPP*, pages 147–156, 2013.

[69] NVIDIA. Cuda software development toolkit v4.2. https://developer.nvidia.com/cuda-toolkit-42-archive.

[70] NVIDIA. Nvidia's next generation cuda computer architecture: Fermi.

[71] NVIDIA. Nvidia's next generation cuda compute architecture: Kepler gk110, 2012. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[72] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU Concurrency with Elastic Kernels. In *ASPLOS*, 2013.

[73] E. Petrank and D. Rawitz. The hardness of cache conscious data placement. In *Proceedings of ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 2002.

[74] T. Preis, P. Virnau, W. Paul, and J. J. Schneider. Gpu accelerated monte carlo simulation of the 2d and 3d ising model. *J. Comput. Phys.*, 228(12):4468–4477, July 2009.

[75] NVIDIA Parallel Thread Execution. http://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[76] V. Ravi, W. Ma, D. Chiu, and G. Agrawal. compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, 2010.

[77] M. Rhu, M. Sullivan, J. Leng, and M. Erez. A locality-aware memory hierarchy for energy-efficient gpu architectures. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 86–98, New York, NY, USA, 2013. ACM.

[78] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-conscious wavefront scheduling. In *MICRO*, pages 72–83, 2012.

[79] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Divergence-aware warp scheduling. In *MICRO*, pages 99–110, 2013.

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

[81] X. Shen, Y. Gao, C. Ding, and R. Archambault. Lightweight reference affinity analysis. In *ICS*, Cambridge, MA, June 2005.

[82] 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.

[83] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of International Symposium on Computer Architecture*, pages 336–349, San Diego, CA, June 2003.

[84] J. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W. Hwu. Efficient compilation of fine-grained spmd-threaded programs for multicore cpus. In *CGO '10: Proceedings of the International Symposium on Code Generation and Optimization*, 2010.

[85] M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *PLDI*, 2003.

[86] I.-J. Sung, J. A. Stratton, and W. mei W. Hwu. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *PACT*, pages 513–522, 2010.

[87] D. Tarjan, J. Meng, and K. Skadron. Increasing memory miss tolerance for simd cores. In *SC*, 2009.

[88] N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading pentium 4 processor. In *PACT*, 2003.

[89] S. Tzeng, A. Patney, and J. D. Owens. Task management for irregular-parallel workloads on the gpu. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, 2010.

[90] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, Jan. 2013.

[91] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press.

[92] B. Wu, E. Zhang, and X. Shen. Enhancing data locality for dynamic simulations through asynchronous data transformations and adaptive control. In *PACT*, 2011.

[93] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. In *PPoPP*, 2013.

[94] R. Wu, B. Zhang, and M. Hsu. Clustering billions of data points using gpus. In *Proceedings of the Combined Workshops on UnConventional High Performance Computing Workshop Plus Memory Access Workshop*, UCHPC-MAW '09, pages 1–6, New York, NY, USA, 2009. ACM.

[95] S. Xiao and W. chun Feng. Inter-block gpu communication via fast barrier synchronization. In *IPDPS*, 2010.

[96] Y. Yan, X. Zhang, and Z. Zhang. Cacheminer: A runtime approach to exploit cache locality on *smp*. *IEEE Transactions on Parallel Distributed Systems*, 11(4):357–374, 2000.

[97] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A gpgpu compiler for memory optimization and parallelism management. In *PLDI*, 2010.

[98] E. Zhang. *Dynamic Optimizations for Irregular Applications on Many-Core Architectures*. PhD thesis, Computer Science Dept., The College of William and Mary, July 2012.

[99] E. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *ASPLOS*, 2011.

[100] E. Z. Zhang, Y. Jiang, Z. Guo, and X. Shen. Streamlining gpu applications on the fly. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, pages 115–125, 2010.

[101] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal. Grace: A low-overhead mechanism for detecting data races in gpu programs. In *PPoPP*, 2011.

[102] J. Zhong and B. He. Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. *CoRR*, abs/1303.5164, 2013.

[103] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *PLDI*, 2004.