On-the-Fly Elimination of Dynamic Irregularities for GPU Computing

Eddy Z. Zhang Yunlian Jiang Ziyu Guo Kai Tian Xipeng Shen

Computer Science Department
The College of William and Mary, Williamsburg, VA, USA
{eddy,jiang,guoziyu,ktian,xshen}@cs.wm.edu

Abstract

The power-efficient massively parallel Graphics Processing Units (GPUs) have become increasingly influential for general-purpose computing over the past few years. However, their efficiency is sensitive to dynamic irregular memory references and control flows in an application. Experiments have shown great performance gains when these irregularities are removed. But it remains an open question how to achieve those gains through software approaches on modern GPUs.

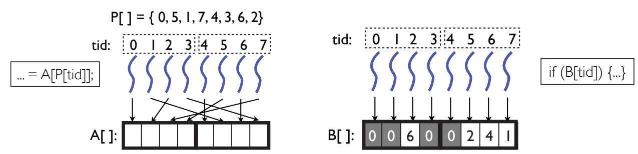
This paper presents a systematic exploration to tackle dynamic irregularities in both control flows and memory references. It reveals some properties of dynamic irregularities in both control flows and memory references, their interactions, and their relations with program data and threads. It describes several heuristics-based algorithms and runtime adaptation techniques for effectively removing dynamic irregularities through data reordering and job swapping. It presents a framework, *G-Streamline*, as a unified software solution to dynamic irregularities in GPU computing. G-Streamline has several distinctive properties. It is a pure software solution and works on the fly, requiring no hardware extensions or offline profiling. It treats both types of irregularities at the same time in a holistic fashion, maximizing the whole-program performance by resolving conflicts among optimizations. Its optimization overhead is largely transparent to GPU kernel executions, jeopardizing no basic efficiency of the GPU application. Finally, it is robust to the presence of various complexities in GPU applications. Experiments show that G-Streamline is effective in reducing dynamic irregularities in GPU computing, producing speedups between 1.07 and 2.5 for a variety of applications.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—optimization, compilers

General Terms Performance,Experimentation

Keywords GPGPU, Thread divergence, Memory coalescing, Thread-data remapping, CPU-GPU pipelining, Data transformation

ASPLOS'11, March 5–11, 2011, Newport Beach, California, USA.



(a) Irregular memory reference (b) Irregular control flow

Figure 1. Examples of dynamic irregularities (warp size=4; segment size=4). Graph (a) shows that inferior mappings between threads and data locations cause more memory transactions than necessary; graph (b) shows that inferior mappings between threads and data values cause threads in the same warp diverge on the condition.

1. Introduction

Recent several years have seen a quick adoption of Graphic Processing Units (GPU) in general-purpose computing, thanks to their tremendous computing power, and favorable cost effectiveness and energy efficiency. These appealing properties come from the massively parallel architecture of GPU, which, unfortunately, entails a major weakness of GPU: the high sensitivity of their throughput to the presence of irregularities in an application.

The massive parallelism of GPU is embodied by the equipment of a number of streaming multiprocessors (SM), with each containing dozens of cores. Correspondingly, a typical application written in GPU programming models (e.g., CUDA [14] from NVIDIA) creates thousands of parallel threads running on GPU. Each thread has a unique ID, *tid*. These threads are organized into warps[1]. Threads in one warp are assigned to a single SM, and proceed in an SIMD (Single Instruction Multiple Data) fashion. As a result, hundreds of threads may be actively running on a GPU at the same time. Parallel execution of such a large number of threads may well exploit the tremendous computing power of GPU, but not for irregular computations.

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

Before explaining irregular data references, we introduce the properties of GPU memory access. (Without noting, "memory" refers to GPU off-chip global memory.) In a modern GPU device (e.g., NVIDIA Tesla C1060, S1070,C2050, S2070), memory is composed of a large number of continuous segments. The size of

[1] This paper uses NVIDIA CUDA terminology.

each segment is a constant[2], denoted as Z. One memory transaction can load or store all data in one memory segment. The accesses by a set of threads at one load or store instruction are coalesced into a single memory transaction, if these threads are within a warp and meanwhile the words accessed by them lie in a single memory segment. An *irregular reference* refers to a load or store instruction, at which, the data requested by a warp happens to lie on multiple memory segments, causing more (up to a factor of W; W for warp size) memory transactions than necessary. Because a memory transaction incurs latency of hundreds of cycles, irregular references often degrade the effective throughput of GPU significantly.

A special class of irregular data references is *dynamic irregular references*, referring to irregular references whose memory access patterns are unknown (or hard to know) until execution time. Figure 1 (a) shows an example. The memory access pattern of "A[P[tid]]" is determined by the runtime values of the elements in array P, whose content causes an irregular mapping between threads and the locations of the requested data, resulting in four memory transactions in total, twice of the minimum. Being dynamic, these references are especially hard to tackle, making effective exploitation of GPU difficult for many applications in various domains, including fluid simulation, image reconstruction, dynamic programming, data mining, and so on [13, 18].

Dynamic irregularities also exist in program control flows, causing *thread divergences*. Thread divergences typically happen on a condition statement. When threads in a warp diverge on which branch to take, their parallel execution turns into a serial execution of the threads that take different branches. Figure 1 (b) illustrates such an example. Consider the first warp in the graph. Due to the values of the data mapped to the threads, only thread 2 takes the "if" branch. During the execution of that thread, all the other threads in that warp have to stay idle and wait. Note that because the warp is not completely idle, no other warps are allowed to run on that SM during that time, causing waste of computing resource. Consider a typical case where each warp contains 32 threads. The waste of the SM throughput is up to 96% (31/32). The problem is especially serious for loops. Consider a loop "for (i=0; i¡A[tid]; i++)" in a kernel and A[0] to A[31] are all zero except that A[13]=100. All threads in the warp have to stay idle until thread 13 finishes the 100^{th} iteration.

Dynamic irregularities severely limit the efficiency of GPU computing for many applications. As shown in Figure 2, removing the dynamic irregularities may improve the performance of a set of GPU applications and kernels (detailed in Section 7) by a factor of *1.4* to *5.3*.

There have been some recent explorations on the irregularity issues. Some propose new hardware features [8, 13, 18], others offer software solutions through compiler techniques [3, 4, 11, 20, 21]. Software solutions, being immediately deployable on real GPUs, are the focus of this paper. Previous software solutions mainly concentrate on cases that are amenable to static program analysis. They are not applicable to *dynamic* irregularities, whose patterns remain unknown until execution time. A recent work [22] tackles dynamic irregular control flows, but in a limited setting (as elaborated in Section 8).

Overall, a systematic software solution to address *dynamic* irregularities in GPU computing is yet to be developed. In fact, what remains missing are not just solutions, but more fundamentally, a comprehensive understanding to the problem of irregularity removal itself. For instance, as Figure 1 shows, the two types of irregularities stem from the relations between GPU threads and runtime data values or layouts, but the relations are preliminarily understood. No answers exist to the questions such as what data layouts and thread-data mappings minimize the irregularities, what the computational complexities are for finding desired layouts or mappings, and how they can be effectively approximated.

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

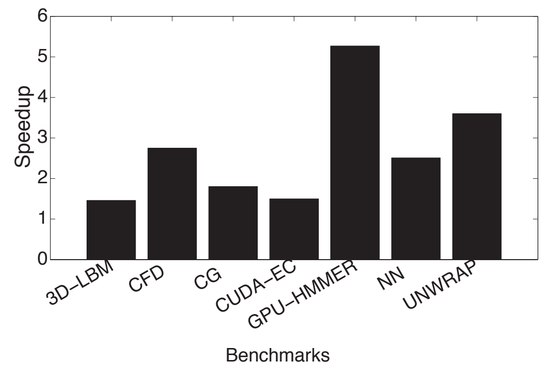


Figure 2. Potential performance improvement when dynamic irregularities are eliminated for applications running on an GPU (Tesla 1060).

Moreover, previous explorations (in software) have treated the two kinds of irregularities separately. But in many real applications, both may exist at the same time and connect with each other—optimizing one may influence the other (e.g., *3dlbm* shown in Section 7). It is important to treat them in a holistic fashion to maximize overall performance.

Overview of This Work In this work, we aim to answer these open questions and contribute a comprehensive, practical software solution to both types of dynamic irregularities. First, we unveil some analytical findings on the inherent properties of irregularities in GPU computing. This includes the interactions between irregular control flows and memory references, the NP-completeness of finding the optimal data layouts and thread-job mappings and a set of heuristics-based algorithms, as well as the relations among dynamic irregularities, program data, and GPU threads. These findings substantially enhance the current understanding of the irregularities. Second, we provide a unified framework, named **G-Streamline**, as a comprehensive solution to both types of dynamic irregularities. G-Streamline has several distinctive properties. It is a pure software solution and works on the fly, requiring no hardware extensions or offline profiling. It treats both types of irregularities at the same time in a holistic fashion, maximizing the whole-program performance by resolving conflicts among optimizations of multiple irregularities of the same or different types. Its optimization overhead is transparent to GPU executions, jeopardizing no basic efficiency of the GPU application. Finally, it is robust to the presence of various complexities in the GPU application, including the concealing of the data involved in condition statements, the overlapping of the data involved in irregular data references.

We build G-Streamline based on a perspective illustrated in Figure 1 (a) and (b): Both irregular memory references and control flows essentially stem from an inferior mapping between threads and data (data locations for the former; data values for the latter). This perspective leads to the basic strategy of G-Streamline for irregularity elimination: enhancing the thread-data mappings on the fly. To make this basic strategy work efficiently, we develop a set of techniques organized in three components as shown in Figure 3.

The component, "transformation" (Section 3), includes techniques for the realization of new thread-data mappings. Its core

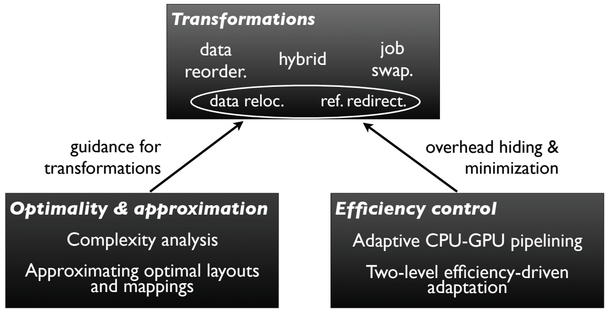


Figure 3. Major components of G-Streamline.

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

The second component, "optimality & approximation" (Section 4), helps meet the first condition by answering a series of open questions on the determination of desirable data layouts and mappings for GPU irregularity removal. It proves that finding the optimal data layouts or thread-data mappings in order to minimize the number of memory transactions is NP-complete. For the minimization of thread divergences, it shows that the problem is NP-complete as well but with respect to the number of conditional branches rather than the number of threads. Based on the theoretical insights, this component provides a heuristics-based algorithm for each type of transformations, enabling the computation of near-optimal data layouts or thread-data mappings. Meanwhile, it offers some guidelines for resolving conflicts among the optimizations of different irregularities.

The third component, "efficiency control" (Section 5), addresses overhead issues. On one hand, because the irregularities are dynamic, optimizations must happen during run time. On the other hand, transformations for irregularity removal are usually expensive due to the data movements and relevant computations involved. To address that tension, the "efficient control" component employs two techniques. First, based on a previous proposal [22], it adopts an adaptive CPU-GPU pipelining scheme to offload most transformations to CPU so that the transformations can happen asynchronously with the GPU kernel execution. The scheme effectively hides transformation overhead from kernel execution, and meanwhile, protects the basic efficiency of the program by automatically shutting down transformations when necessary. Second, it uses a multilevel adaptation scheme to reduce transformation overhead. The first level is on the tuning of individual transformations; the second level is on the selection of different transformation methods, according to their distinctive properties and the runtime scenarios.

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

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

2. Terms and Abstract Forms

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

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

In the following discussions, we use the abstract form "A[P[tid]]" to represent an irregular reference, and "if (B[tid])" to represent an irregular control flow. The arrays "P" and "B" are both conceptual. In real applications, "P" may appear as an actual input array, or results computed from some input arrays (e.g., "A[X[tid]%2+Y[tid]]"), while, "B" may appear as a logical expression on some input arrays. Using these abstract forms gives conveniences to our discussion, but does not affect the generality of the conclusions (elaborated in Section 6).

3. Transformations for Irregularity Removal

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

3.1 Two Basic Transformation Mechanisms

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

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

3.2 Three Transformation Methods

We develop three transformation methods for removing irregular control flows and memory references. Each of them consists of a series of applications of the two basic mechanisms. In the following explanation on how the transformations remove dynamic irreg-

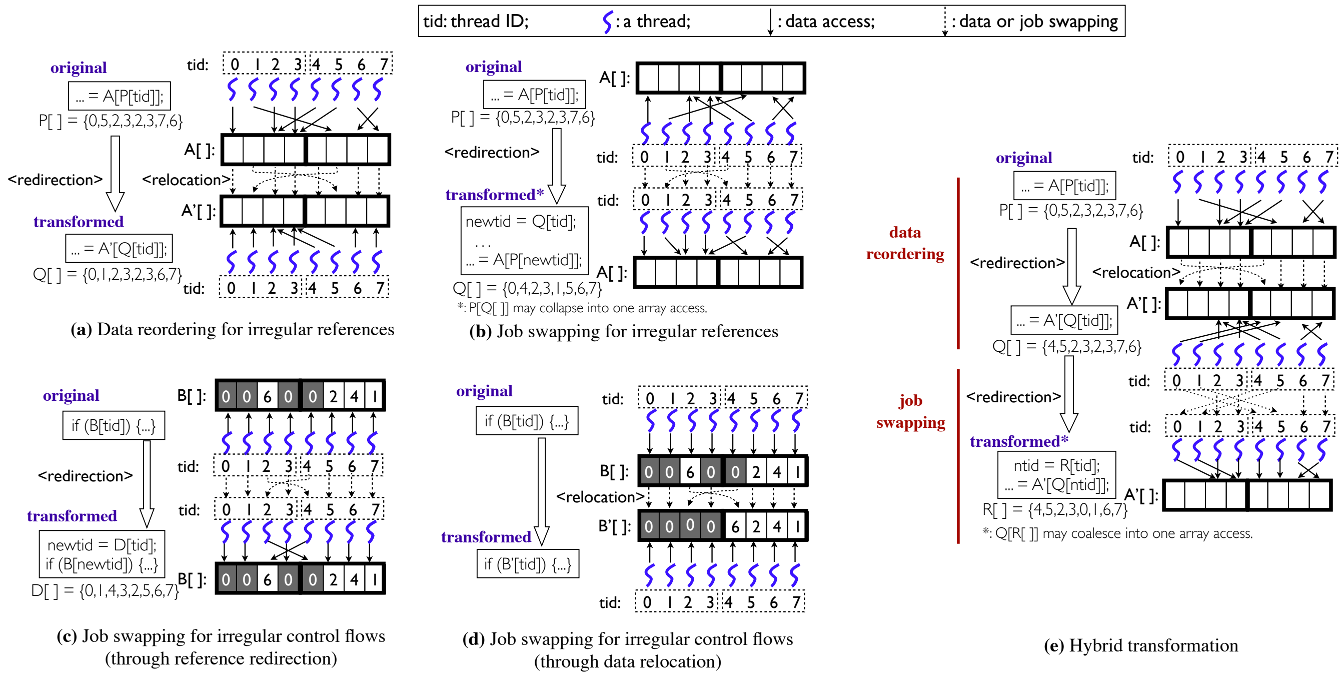


Figure 4. Examples for illustrating the uses of data reordering and job swapping for irregularity removal.

ularities, we assume that the desirable mappings between threads and data (locations or values) are known. Section 4 discusses how to determine those mappings.

3.2.1 Data Reordering

The first strategy is to adjust data locations on memory to create a new order for the elements of an array. Its application involves two steps, as illustrated in Figure 4 (a). In the first step, data relocation creates a new array A' that contains the same set of elements as the original array A does but in a different order. The new order is created based on a desirable mapping (obtained from P as shown in Section 4) between threads and data locations. In our example, originally, the values of the elements in P cause every warp to reference elements of A on two segments (the top half of the graph). The relocation step switches the locations of $A[5]$ and $A[1]$. The second step of the transformation changes accesses to A in the kernel such that each thread accesses the same data element (likely in a different location) as it does in the original program. The boxes in the left part of Figure 4 (a) illustrates the change: $A[P[tid]]$ is replaced with $A[Q[tid]]$, where Q is a newly produced redirection array. After this transformation, all data accessed by the threads in the first warp lie in the first segment; the total needed memory transactions is reduced from four to three. (Section 3.2.3 will show how to reduce it further to the minimum.)

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

3.2.2 Job Swapping

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

As shown in Figure 4 (b), by exchanging the jobs of threads 1 and 4, we make thread 1 access $A[2]$ and thread 4 access $A[5]$. The

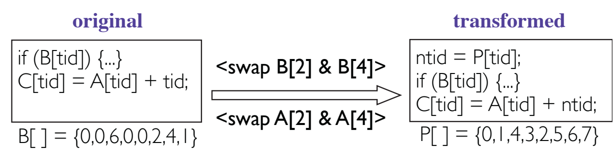


Figure 5. Using data relocation for job swapping faces some complexities.

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

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

Job swapping can be materialized in two ways. Besides through reference redirection as Figures 4 (b) and (c) show, the second way is through data relocation. As shown in Figure 4 (d), when the locations of $B[2]$ and $B[4]$ switch while tid remains unchanged in the kernel, threads 2 and 4 automatically swap their jobs. There are some complexities in applying this job swapping method, exemplified by Figure 5. First, it requires all input arrays in the kernel (e.g., A and B in Figure 5) go through the same data exchanges to maintain the integrity of a job. The incurred data copying may cause large overhead. Second, for this approach to work, it must treat occurrences of tid that are outside array subscripts carefully. For instance, in Figure 5, simply switching $A[2]$ and $A[4]$ on memory would cause the expression "A[tid]+tid" to produce wrong results. A careful treatment to appearances of "tid" that are outside array subscripts can fix the problem, as shown in the transformed code in Figure 5 (where, P is an assistant array created to record the mapping between threads and jobs). Finally, at the end of the kernel, the order of the elements in output arrays (e.g., C in Figure 5) has to be restored (e.g., switch $C[2]$ and $C[4]$) so that the order of elements match with the output of the original program.

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

3.2.3 Hybrid Transformations

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

The second benefit is that combined transformations often lead to greater reduction of memory transactions than each individual transformation does. As shown in Figure 4 (a) and (b), data reordering and job swapping both reduce the needed memory transactions to three for the shown example. Figure 4 (e) shows that a combination of the two transformations may reduce the number of memory transactions to two, the minimum. The rationale for the further reduction is that the reordering step creates a data layout that is more amenable for job swapping to function than the original layout is. On the new layout, two threads in warp one reference two data elements in segment two, and meanwhile, two threads in warp two reference two data elements in segment one. Swapping the jobs of the two pairs of threads ensures that the references by each warp fall into one single segment, hence minimizing the number of needed memory transactions.

3.2.4 Comparisons

Both types of irregularities may benefit from multiple kinds of transformations. We briefly summarize the properties of the various transformations. Section 5 describes the selection scheme adopted in G-Streamline.

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

Thread divergence removal relies mainly on job swapping with data reordering as a follow-up remedy for side effects. Between the two ways to realize job swapping, the redirection-based method has lower overhead than the relocation-based method, as by itself, no data movements are needed. However, that benefit is often offset by its side effect on memory references. On the other hand, the relocation-based method, although having no such side effects, are limited in applicability. Generally, if the data to be moved are accessed by threads in more than one warp, relocation-based job swapping is likely to encounter difficulties. (Consider a modified version of the example in Figure 4 (d), where thread 4 originally accesses B[3] rather than B[4].)

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

4. Determination of Desirable Data Layouts and Mappings

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

4.1 Irregular Memory References

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

In this work, by reducing known NP-complete problems, the *3DM* and the *partition problem* [10], we prove that finding optimal data layouts or thread-data mappings is NP-complete for minimizing the number of memory transactions. For lack of space, we elide the proofs, but describe two heuristics-based solutions, respectively for data reordering and job swapping.

Data Reordering For data reordering, we employ data duplication to circumvent the difficulties in finding optimal data layouts. The idea is simple. At a reference, say $A[P[tid]]$, we create a new copy of A, denoted as A', such that $A'[i] = A[P[i]]$. Then, we use $A'[tid]$ to replace every appearance of $A[P[tid]]$ in the kernel. With this approach, the number of memory transactions at the reference equals the number of thread warps—the optimal is achieved. The main drawback of this approach is space overhead: When n threads reference the same item in A, there would be n copies of the item in A'. When there are irregular references to multiple arrays (or multiple references to one array with different reference patterns, e.g., $A[P[tid]]$ versus $A[Q[tid]]$) in the kernel, the approach creates duplications for each of those arrays (or references), hence possibly causing too much space overhead. Section 5 will show how adaptive controls address this problem.

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

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

4.2 Irregular Control Flows

As Section 3 describes, only job swapping is applicable for removing irregular control flows. This section focuses on reference redirection-based job swapping for its broad applicability. The key to its effectiveness is to find a desirable mapping between threads and jobs.

Through reducing the *partition problem* [10], we prove that finding optimal thread-job mappings (in terms of the total number of thread divergences) for the removal of irregular control flows is NP-complete with respect to K (K is the number of condition statements in a kernel; assuming each has two branches). The proof is elided for lack of space.

Designing heuristics-based algorithms for removing irregular control flows is not a main focus of this work. We extend the algorithms proposed in our previous work [22]. In the prior study, we used path-vector–based job regrouping to handle divergences caused by non-loop condition statements. For a kernel with K condition statements, each job has a corresponding K-dimensional vector (called *path vector*), with each member equaling the boolean value on a condition statement. The prior work uses loop trip-count (i.e., number of iterations) based sorting to treat thread divergences caused by a loop termination condition. It describes no solutions to the scenarios where both kinds of conditions co-exist. We handle such cases by adding one dimension to the path vector for each loop. The values in those dimensions are categorized loop trip-counts. The categorization is through distance-based clustering [9]. For instance, for a kernel with two condition statements and one loop whose iterations among all threads fall into L clusters (i.e., 0, 100-200, 1000-1300, >10000), the path vectors of all threads would be in three dimensions; the final dimension is for the loop, and can have only L possible values, corresponding to the L clusters. After integrating loops into path vectors, we can simply assign jobs having the same path vector values to threads in the same warps.

4.3 Co-Existence of Irregularities

Irregular control flows and irregular memory references co-exist in some kernels. The co-existence may be inherent in the kernel code, or caused by optimizations as already exemplified in Figure 4 (c). As the optimal data layouts or thread-job mappings may differ for the two types of irregularities, the co-existence imposes further challenges to irregularity removal.

G-Streamline circumvents the problem based on the following observation: Even though job swapping affects both control flows and memory references for a thread, data reordering affects only memory references. The corresponding strategy taken by G-Streamline is to first treat irregular control flows using the approach described in the previous sub-section, and then apply data reordering to handle all irregular memory references, including those newly introduced by the treatments to irregular control flows. The handling of irregular memory references does not jeopardize the optimized control flows.

5. Adaptive Efficiency Control

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

G-Streamline overcomes the difficulty through a CPU-GPU pipelining scheme, a set of overhead reduction methods, and a suite of runtime adaptive control. These techniques together ensure that the optimizations do not slow down the program in the worst case, and meanwhile, maximize optimization benefits in various scenarios by overlapping transformations with kernel executions, circumventing dependences, and adaptively adjusting transformation parameters. Our description starts with the basic CPU-GPU pipelining—the underlying vehicle supporting the various transformations.

5.1 Basic CPU-GPU Pipelining for Overhead Hiding

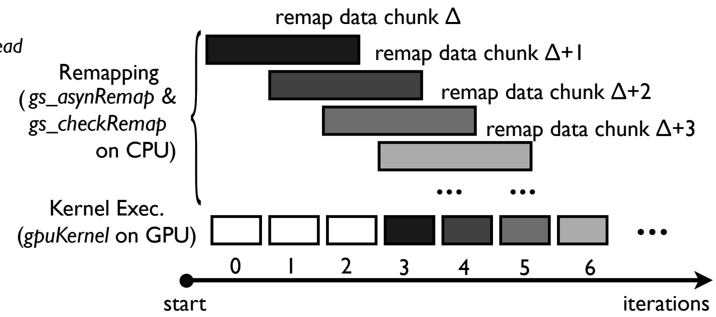
The basic idea of the CPU-GPU pipelining is to make transformations happen asynchronously on CPU when GPU kernels are running. We first explain how the pipelining works in a setting where the main body of the original program is a loop. Each iteration of the loop invokes a GPU kernel to process one chunk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

Figure 6 shows an example use of the CPU-GPU pipelining. The CPU part of the original program is in normal font in Figure 6 (a). Each iteration of its central loop invokes *gpuKernel* to make the GPU process one chunk of the data. The italic-font lines are inserted code to enable the pipelined thread-data remapping. All functions with the prefix "gs_" are part of the G-Streamline library. Consider that the execution of the ith iteration of the loop. At the invocation of "gs_asynRemap ( )", the main CPU thread wakes up an assistant CPU thread. While the assistant thread does thread-data remapping for the chunk of data that is going to be used in iteration $(i+\Delta)$, the main thread moves on to process the ith chunk of data. It first checks whether the G-Streamline transformation (started in the $(i-\Delta)$th iteration) for the current iteration is already done. If so (i.e., *cpucpyDone[i]* is true), it invokes the optimized GPU kernel; otherwise, it uses the original kernel. While the GPU executes the kernel, the main CPU thread moves on to "gs_checkRemap $(i+1)$" (pseudo code in the box) to copy the transformed $(i+1)th$ chunk of data from host to GPU. This copying is conditional: The first while loop in "gs_checkRemap ()" ensures that the copying starts only if the transformation completes before the ith GPU kernel invocation finishes. The second "while" loop ensures that the main thread moves on normally without waiting for the data copying to finish if the ith GPU kernel invocation has completed. These two "while" loops together guarantee that the transformation and

```
pData = & data;
for (i=0; i<N/S; i++) {...
    gs_asynRemap (pData+Δ*S); // remap by assist. CPU thread
    if (cpucpyDone[i])
        gpuKernel_gs_opt <<<... ...>>> (pData, ...);
    else
        gpuKernel_org <<<... ...>>> (pData, ...);
    gs_checkRemap (i+1); ...}

procedure gs_checkRemap (j) {
    if (j<Δ) return; // no remapping for this iteration
    while ( (!gpuDone[j-1]) AND (!cpuoptDone[j-Δ]) );
    if (cpuoptDone[j-Δ]) {
        gs_dataCpy (j-Δ); // asynchronous copy to GPU
        while ( (!gpuDone[j-1]) AND (!cpucpyDone[j-Δ]) );
    }
}
```

(a) CPU part of an example program with G-Streamline code (italic) inserted.



(b) Pipeline when the depth Δ is 3

Figure 6. An example illustrating the (simplified) use of CPU-GPU pipelining to hide the overhead in thread-data remapping transformations. The code in the bottom box is part of the G-Streamline library.

associated data copying cause no delay to the program execution even in the worst case.

The status arrays, *gpuDone*, *cpuoptDone*, and *cpucpyDone* in Figure 6, are conceptual. The "cudastreamquery" in CUDA API is actually used for checking the GPU kernel status.

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

5.2 Dependence and Kernel Splitting

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

We introduce a technique called *kernel splitting* to solve the problem. The idea is to split the execution of a GPU kernel into two by duplicating the kernel call and distributing the tasks. Figure 7 shows such an example. In the new program, the invocation of the original kernel *"gpuKernel_org"* is replaced with *gpuKernel_org_sub* and *gpuKernel_opt_sub*. The invocation of the function *gpuKernel_org_sub* behaves the same as the original, but completes only the first $(1-r)$ portion of the data processed by the original kernel (i.e., the tasks conducted by the first $(1-r)$ portion of the original GPU threads), while the invocation of function *gpuKernel_opt_sub* completes the remaining tasks. When GPU is executing *gpuKernel_org_sub*, a CPU assistant thread does G-Streamline transformations for the data to be used in *gpuKernel_opt_sub*. Therefore, with the kernel execution split into two, the CPU-GPU pipelining becomes feasible even in the presence of dependences. The rate r is called *optimization ratio*, the determination of which is discussed in Section 5.4.

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

```
gpuKernel_org<<<...>>>(pData,...);
        | split
gpuKernel_org_sub<<<...>>>(pData,0, (1-r)*len, ...);
gpuKernel_org_sub<<<...>>>(pData,(1-r)*len+1, len, ...);
```

Figure 7. Kernel splitting makes CPU-GPU pipelining remain feasible despite loop-carried dependences.

5.3 Approximation and Overlapping

In some cases, the overhead of a full transformation is so large that even the pipelining cannot completely hide the overhead. Approximations are necessary to trade optimization quality for efficiency. The partial transformation mentioned in the previous subsection is one example of such approximations. By only transforming part of the data set that is going to be used in an iteration, the technique reduces transformation time. Even though that technique is described for addressing loop-carried dependences, partial transformation is apparently applicable to all settings regardless of the presence of dependences.

For the elimination of control flow irregularities, we adopt the label-assign-move (LAM) algorithm described in our previous work [22]. The algorithm avoids unnecessary data movements by marking data with a number of class labels and making only necessary switching of data elements such that same classes of data locate adjacently.

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

5.4 Adaptive Control

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

Coarse-Grained Adaptation The first level of adaptation exists in the CPU-GPU pipelining. As Section 5.1 already shows, a transformation shuts down automatically if it runs too slow, and the main CPU thread moves on to the next iteration regardless of whether the transformation finishes. This level of adaptation guarantees the basic efficiency of the program execution.

The second level of adaptation selects appropriate transformation method to use. Recall that irregular reference removal can benefit from different types of transformations. The implementations of these transformations in G-Streamline show the following properties. Data reordering has the largest space overhead and medium time overhead, but is able to remove all irregular memory references (with data duplication). Job swapping has the smallest overhead in both space and time, but has limited effectiveness and applicability. The strategy in G-Streamline is to use data reordering as the first choice. If its space overhead is intolerable, G-Streamline switches to job swapping. To enable this level of adaptation, multiple copies of the kernel code would need to be created, with each containing the code changes needed for the corresponding transformation. This level of adaptation is optional in the use of G-Streamline.

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

The pipeline depth Δ (Section 5.1) determines the time budget for a transformation to finish. In our implementation, we fix it as 1 but allow multiple threads (depending on the number of available CPU cores) to transform for one chunk of data in parallel. This implementation simplifies thread management.

The number of data segments s (Section 5.3) influences the overlapping between transformation and data copying. Its value is 1 by default. In the initial several iterations, if G-Streamline finds that the transformation overhead always exceeds the kernel running time despite what values r takes, it increases this parameter to 5, a value working reasonably well for most benchmarks in our experiments.

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

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

  Figure 8 shows the adjustment policy in G-Streamline. As the right part of the flow chart indicates, r increases its value on two (rather than one) consecutive successes to avoid the problem mentioned in the previous paragraph. An additional condition for the increase is that the value of r has never been decreased. If r has been decreased, two consecutive successes means that the appropriate value of r has been found (further increase can only cause failures, and further decrease produces less optimization benefits), and the adjustment ends. All future iterations use that r value.

In the case that kernel splitting is needed for dependences carried by the central loop, the dynamic adjustment of r is the same as shown in Figure 8 except that the top two boxes on the right are removed. It is due to the fact that the transformed data are used in the current iteration.

In the case that there is no central loop (e.g., *cuda-ec* in Section 7), The kernel tasks are split into three parts, executed by three kernel calls. The first part contains 10% of all. During its execution, the CPU transforms 10% of data. After that, the CPU thread uses the measured kernel time T_{org} and the transformation time T_{tr} to estimate what portion (α) of the remaining 90% tasks should run in the second kernel call so that the transformation for the remaining $(1-\alpha)*90\%$ tasks can finish before the finish of the second kernel call. The calculation is $\alpha = T_{tr}/(T_{tr} + T_{org})$. The G-Streamline then optimizes for the remaining $(1 - \alpha) * 90\%$ tasks while the second kernel call is working on the $\alpha * 90\%$ tasks. If the second kernel call still finishes early, the transformation is terminated immediately. Otherwise, the third kernel call uses the optimized data to gain speedups.

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

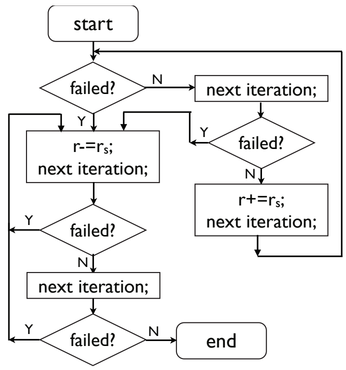


Figure 8. Dynamic adjustment for optimization ratio.

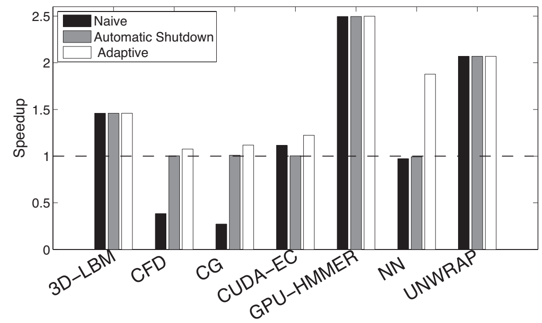


Figure 9. Speedup from thread-data remapping.

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

6. Usage and Other Issues

G-Streamline is in form of a library, written in C++ and Pthreads, consisting of functions for various transformations (including the heuristics-based algorithms) described in this paper, along with the functions for enabling the CPU-GPU pipelining and the adaptation schemes. To activate the pipelined transformations, users need to insert several function calls into the CPU code that encloses GPU kernel invocations. Some minor code changes are necessary to GPU kernels, such as the changes to array reference subscripts as shown in Figure 4. Currently, the changes are done manually.

Discussions in this paper have been based on the abstract forms of irregular references ("A[P[tid]]") and condition statements ("if (B[tid])") defined in Section 2. In our experiments, we find that for most applications, "P" and "B" are either some input arrays or results derived by a simple calculation on input arrays. In these cases, their values are easy for G-Streamline to obtain through a simple pre-computation on input arrays before applying the transformations. But in few kernels, the calculations of "P" and "B" are complex. To handle such cases, G-Streamline provides an interface for programmers to provide functions for the attainment of "P" and "B". For efficiency, the function can produce approximated values. The calculation of "P" and "B" is part of the transformation process in G-Streamline, and hence can be hidden by the CPU-GPU pipelining scheme and jeopardizes no basic efficiency of the application.

7. Evaluation

We evaluate the effectiveness of G-Streamline on a set of benchmarks shown in Table 1. We select them because they contain some non-trivial dynamic irregularities. The benchmarks come from some real applications [17, 23] and some recently released GPU benchmark collections, including Rodinia [5] and NVIDIA Tesla Bio [19]. One exception is *cg*, a kernel derived from an OpenMP program in the NAS suite [2]. Including it is for a direct comparison with a prior study [11] that has optimized the program intensively.

The seven benchmarks cover a variety of domains, and have different numbers and types of irregularities. The program *3dlbm* contain both diverging branches and irregular memory references. Four of the others have irregular memory references, and the other two contains only thread divergences. Together they make a mixed set for the evaluation of not only the various transformations in G-Streamline but also its adaptation schemes. The original implementation of these programs are in CUDA. Previous documents have shown that they have gone through carefully tuning and outperformed their CPU counterparts substantially (e.g., 10–467x for 3dlbm [23], 20x for cuda-ec [19],9–30x for cfd [5]). The inputs to these programs are shown in Table 1, some of which (e.g., the input to *cfd*) are directly obtained from the authors of the benchmarks as the ones coming with the benchmark suite are too small for experiments and typical practical uses.

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

7.1 Results Overview

Figure 9 reports the speedups of the optimized kernels in the seven benchmarks. The baseline is the execution times of the original kernels; bars higher than 1 means speedup, and slowdown otherwise.

Each program has three bars, respectively corresponding to the performance when the transformations are applied with no adaptation control, with first-level control (i.e., automatic shutdown when transformations last too long), and with all adaptions. The first group of bars indicate that the brute-force application of the transformations, although leading to significant speedup to three programs, cause drastic slowdown to two programs. The first-level adaptive control (automatic shutdown) successfully prevents the slowdown, while the other levels of adaptations yield further substantial performance improvement to most programs. (The reason for the first-level adaptation to throttle speedup of *cuda-ec* is shown in Section 7.4.) The benefits of the optimizations are confirmed by the significant reduction of divergences and memory transactions reported by the CUDA profiler, shown in Table 2. Overall, the optimizations yield speedups between 1.07 and 2.5, demonstrating the effectiveness of G-Streamline for exerting GPU power for irregular computations. We acknowledge that compared to the data shown in Figure 2, some of the results are substantially below the full potential. It is mainly due to the dependences across central loops, transformation overhead, and approximation errors. It indicates possible opportunities for further refinement of G-Streamline.

The software rewriting overhead mainly consists of insertion of G-Streamline library calls for data reordering and threads swapping, customized condition computation functions, and the transformed GPU kernels that optimizes data access patterns and control divergence. Table 1 reports the software overhead in terms of the number of lines of inserted code. For most programs, the majority of the inserted code is the duplication of the original code because the new GPU kernels are typically the same as the original except that the thread IDs or array reference IDs are replaced with new IDs. The numbers of lines of newly created code are shown

Table 1. Benchmarks and some dynamically determined optimization parameters

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

Benchmark	Source	Description	Irreg.	Input	sloc	added sloc		r	s
						all	new		
3dlbm	real app. [23]	partial diff. equation solver	div & mem	32X32X32 lattice	1.7k	200	50	1	1
cfd	Rodinia [5]	grid finite volume solver	mem	800k mesh	0.6k	550	200	0.37	5
cg	NAS (rewritten [11])	conjugate gradient method	mem	75k array	1.2k	250	200	0.3	5
cuda-ec	Tesla Bio [19]	sequence error correction	div	1.1M DNA seq.	2.5k	900	150	0.65	1
gpu-hmmer	Tesla Bio [19]	protein sequence alignment	div	0.2M protein seq.	28k	350	100	1	1
nn	Rodinia [5]	nearest neighbor cluster	mem	150M data	0.3k	210	150	0.7	1
unwrap	real app. [17]	3-D image reconstruction	mem	512x512 images	1.4k	100	70	1	1

Table 2. Numbers of thread divergences and memory transactions on one GPU SM reported by hardware performance counters

opt (div): divergence eliminated; opt (div & mem): memory references and divergences optimized.

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

by the "new" column in the table. We acknowledge that the current design of the G-Streamline interface can be further improved to enable more concise expression. Moreover, compiler transformation may further simplify the code changes.

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

7.2 Programs with Independent Loops

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

UNWRAP The program, *unwrap*, is for reconstructing 3-D models of biological particles from 2-D microscope photos [17]. Each iteration of the central loop invokes a GPU kernel to transform an image from the Cartesian coordinate system to the Polar coordinate system. In doing so, it accesses the data points in a series of concentric circles, from the image center to the outskirt. The reference patterns lead to inefficient memory accesses.

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

As Table 2 shows, the transformation reduces the numbers of memory transactions by over 77%. The following table explains the reduction by showing the breakdown of different sizes of memory transactions. (In the GPU, data can be accessed in 32B-, 64B-, or 128B- segments with the same time overhead.) After putting data accessed by the same warp close on memory, the optimization aggregates many small transactions into some large ones, hence reducing the total number of transactions significantly, cutting execution time by half.

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

NN The nearest neighbor application, *nn*, finds the k-nearest neighbors from an unstructured data set. Each iteration of the central loop reads in a set of records, computes the Euclidean distances from the target latitude and longitude. The master thread evaluates and updates the k nearest neighbors. We optimized the read accesses to the unstructured data set through data reordering. We used both the distance computation kernel and the data transfer from host to device to hide the transformation overhead. As Figure 9 shows, the overhead for optimizing the whole kernel run can't be completely hidden. Using the adaptive scheme, we were able to achieve a speedup of about 1.8 with the automatically selected optimization ratio equaling 0.7.

3DLBM The program, *3dlbm*, is a partial differential equation solver based on the lattice Boltzmann model (LBM) [23]. It contains both divergences and irregular memory references. Thread divergences mainly come from conditional node updates. The memory reference patterns in the kernel depend on the dimensions of the GPU thread blocks. A previous study [22] has showed up to 47% speedup. But it concentrates on the removal of thread divergences and uses ad-hoc transformations to resolve memory issues. In this work, we apply G-Streamline to the program and achieves a similar degree of speedup. The follow-up data reordering transformation successfully cuts both the newly introduced irregular references and the originally existing ones. The number of memory transactions reduces by over 74%. Both analysis and transformations happen asynchronously outside the main loop because the order does not need to change across iterations.

GPU-HMMER The application *gpu-hmmer* is a GPU-based implementation of the HMMER protein sequence analysis suite, which is a suite of programs that uses Hidden Markov Models (HMMs) to describe the profile of a multiple sequence alignment. Thread divergences due to the different lengths of protein sequences impairs the program performance. We remove the divergence by job swapping. We replace the original thread-id with reordered thread-id except that the thread-ids used in read/write accesses of intermediate result arrays remain unchanged because it hurts no correctness of the program and keeps memory accesses regular. As Table 2 shows, the elimination of thread divergences happen to reduce the number of memory transactions as well, indicating that as threads work in a more coordinate way, they fetch data more efficiently than before. We obtain a speedup of 2.5. The thread-data remapping overhead is completely hidden by the kernel executions in the central loop.

7.3 Programs with Loop-Carried Dependences

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

CFD The program, *cfd* is an unstructured grid finite volume solver for three-dimensional Euler equations for compressible flow [5]. The inefficient memory references come from the reading of the features of neighboring elements of a node in the unstructured grid of the solver.

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

CG The program, *cg*, is a Conjugate Gradient benchmark [2]. Lee and others [11] have shown that careful optimizations are necessary when translating *cg* from an OpenMP version to GPU code because of its irregular memory references. They demonstrate that static compiler-based techniques may coalesce some static irregular references in its kernel and achieve substantial performance improvement. But they give no solution to the dynamic irregular references to a vector in its sparse matrix vector multiplication kernel. The vector is read and modified in each iteration, causing loop-carried dependence.

G-Streamline tackles those remaining irregular references by applying data reordering transformation to the vector. The analysis step resides outside of the main loop as the suitable data order does not vary. But the transformation step is in each iteration. G-Streamline decides on 30% data transformation, and produces 12% further performance improvement over the version optimized through the previous technique [11].

7.4 Program with No Central Loop

The program, *cuda-ec*, is a parallel error correction tool for short DNA sequence reads. It contains no central loop, but several kernel function calls. We optimize the main kernel *fix_errors1* by removing divergence through job swapping. As Figure 9 shows, the simple application of optimizations without adaptations yields speedup of 1.12. The simple adaptive scheme with automatic shutdown turns off optimizations by default because it cannot tell whether the transformation is beneficial for lack of central loops. G-Streamline, equipped with the complete adaptive control, is able to use the split kernels to estimate optimization ratio (following the scheme described at the end of Section 5.4) such that the transformations can overlap with partial kernel executions. The estimated optimization ratio is 0.65, yielding a speedup of 1.22.

8. Related Work

Several previous studies have proposed hardware extensions for reducing the influence of irregular memory references or control flows on GPU program performance. Meng and others [13] introduce dynamic warp subdivision to divide a warp so that diverging threads can execute in an interleaving manner. Tarjan and others [18] propose adaptive slip to allow a subset of threads to continue while other threads in the same warp are waiting for memory. Fung and others [8] try to reduce thread divergences through dynamic warp formation. These hardware approaches have shown promising simulation results. As a pure software solution, our approaches are immediately deployable on current real GPU systems.

In software optimizations, the work closest to this study is our previous study on thread divergence removal [22]. We show that some thread divergences can be removed through runtime optimizations with the support of a CPU-GPU pipeline scheme. This work is enlightened by that study, but differs from it in several major aspects. First, the previous study tackles only thread divergences, while this study shows that it is important to treat thread divergences with irregular memory references at the same time because of their strong connections. We provide a systematic way to tackle both types of irregularities in a holistic manner, including novel techniques stimulated by the distinctive properties of dynamic irregular memory references on GPU. Second, we contribute some in-depth understanding of the inherent properties of irregularity removal, including the NP-completeness of the problems and the approximation algorithms. They substantially enhance current understanding of GPU irregularity removal. Third, even though the previous study has used reference redirection and data relocation for removing thread divergences, our work reveals the full spectrum of transformations that can be constructed from the two basic mechanisms, and uncovers the properties of each type of transformations. Finally, our work develops some novel efficiency-driven adaptations. Together, these innovations advance state of the art of GPU irregularity removal in both theoretical and empirical aspects.

Another work on thread divergences is from Carrillo and others [4]. They use loop splitting and branch splitting in order to alleviate register pressure caused by diverging branches, rather than to reduce thread divergences.

There have been a number of studies on optimizing GPU memory references. The compiler by Yang and others [21] optimizes memory references that are amenable for static transformations. Lee and others [11] show the capability of an openMP-to-CUDA compiler for optimizing memory references during the translation process. Baskaran and others [3] use a polyhedral compiler model to optimize affine memory references in regular loops. Ueng and others [20] show the use of annotations for optimize memory references through shared memory. Ryoo and others [16] demonstrate the potential of certain manual transformations.

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

An orthogonal direction for enhancing GPU program performance is through auto-tuning tools [12, 15]. The combination of dynamic irregularity removal and auto-tuning may offer some special optimization opportunities.

In CPU program optimizations, data relocation and reference redirection have been exploited for improving data locality and hence cache and TLB usage (e.g., [1, 6, 7]). As a massively parallel architecture, GPU display different memory access properties from CPU, triggering the new set of innovations in this paper on both complexity analysis and transformation techniques.

9. Conclusion

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

Acknowledgments

We thank Mary Hall for her help during the preparation of the final version of the paper. The computational complexity analysis in the paper benefits from discussions with Weizhen Mao. We owe the anonymous reviewers our gratitude for their helpful suggestions on the paper. We thank Andrew Corrigan for kindly providing us new input set for *cfd*. This material is based upon work supported by the National Science Foundation under Grant No. 0720499, 0811791 and 0954015, and IBM CAS Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or IBM.

References

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

[2] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report 103863, NASA, July 1993.

[3] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for GPGPUs. In *ICS*, 2008.

[4] S. Carrillo, J. Siegel, and X. Li. A control-structure splitting optimization for gpgpu. In *CF*, 2009.

[5] 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.

[6] T. M. Chilimbi and R. Shaham. Cache-conscious coallocation of hot data streams. In *PLDI*, 2006.

[7] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *JPDC*, 2004.

[8] W. Fung, I. Sham, G. Yuan, and T. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *MICRO*, 2007.

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

[10] D. S. Hochbaum. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1995.

[11] S. Lee, S. Min, and R. Eigenmann. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In *PPoPP*, 2009.

[12] 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.

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

[14] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, pages 40–53, Mar./Apr. 2008.

[15] G. Rudy, C. Chen, M. Hall, M. Khan, and J. Chame. Using a programming language interface to describe gpgpu optimization and code generation. 2010.

[16] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP*, 2008.

[17] G. Tan, Z. Guo, M. Chen, and D. Meng. Single-particle 3d reconstruction from cryo-electron microscopy images on gpu. In *ICS*, 2009.

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

[19] TeslaBio. NVIDIA Tesla Bio Workbench. http://www.nvidia.com/object/tesla_bio_workbench.html.

[20] S. Ueng, S. Baghsorkhi, M. Lathara, and W. Hwu. Cuda-lite: Reducing gpu programming complexity. In *LCPC*, 2008.

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

[22] E. Z. Zhang, Y. Jiang, Z. Guo, and X. Shen. Streamlining gpu applications on the fly. In *ICS*, 2010.

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