Streamlining GPU Applications On the Fly

—Thread Divergence Elimination through Runtime Thread-Data Remapping

Eddy Z. Zhang Yunlian Jiang Ziyu Guo Xipeng Shen
Computer Science Department
The College of William and Mary, Williamsburg, VA, USA 23185
{eddy,jiang,guoziyu,xshen}@cs.wm.edu

ABSTRACT

Because of their tremendous computing power and remarkable cost efficiency, GPUs (graphic processing unit) have quickly emerged as a kind of influential platform for high performance computing. However, as GPUs are designed for massive data-parallel computing, their performance is subject to the presence of condition statements in a GPU application. On a conditional branch where threads diverge in which path to take, the threads taking different paths have to run serially. Such divergences often cause serious performance degradations, impairing the adoption of GPU for many applications that contain non-trivial branches or certain types of loops.

This paper presents a systematic investigation in the employment of runtime thread-data remapping for solving that problem. It introduces an abstract form of GPU applications, based on which, it describes the use of reference redirection and data layout transformation for remapping data and threads to minimize thread divergences. It discusses the major challenges for practical deployment of the remapping techniques, most notably, the conflict between the large remapping overhead and the need for the remapping to happen on the fly because of the dependence of thread divergences on runtime values. It offers a solution to the challenge by proposing a CPU-GPU pipelining scheme and a label-assign-move (LAM) algorithm to virtually hide all the remapping overhead. At the end, it reports significant performance improvement produced by the remapping for a set of GPU applications, demonstrating the potential of the techniques for streamlining GPU applications on the fly.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*optimization, compilers*

General Terms

Performance,Experimentation

Keywords

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

1. INTRODUCTION

Because of their remarkable computing power and cost efficiency, GPUs (Graphics Processing Units) have emerged as a kind of influential platform for high performance computing [2, 7, 12–15].

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

Thread divergence often causes serious performance degradations. Figure 1 shows a piece of code adapted from a program named *gafort* that performs a genetic algorithm. Suppose the first 32 elements in $r1$ are all even numbers except $r1[13]$. When the first warp encounters the "if" statement in the code, only thread 13 passes through the check and conducts *compute1*, while all the other 31 threads are idle and waiting. Note that because the warp is not completely idle, no other warps are allowed to run on that SM during that time. So at most 1/32 computing power of the SM is being used. The similar problem exists on the "for" loop in Figure 1. Suppose that the first 32 elements in $r2$ are all as large as *nchrome* except that $r2[4]$ is 0; then all the 31 threads have to stay idle and wait until thread 4 finishes its *nchrome* (which could be very large) iterations of *compute2*, causing substantial waste of computing power. In reality, we observe up to 1.47 speedup when thread divergences are removed (as shown in Section 5), which echos the potential observed in previous studies [3, 8].

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

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

Figure 1: A piece of code adapted from *gafort*, a program implementing a genetic algorithm. Both the "if" statement and the "for" loop may cause thread divergence. (*tid* is the sequential number of the current thread.)

As a side effect of the architectural support of GPUs for massive data parallelism, this divergence problem exists in virtually all types and generations of modern GPUs. It impairs the adoption of GPU for many applications that contain non-trivial condition statements. There have been limited solutions proposed, among which, some [3] aim at reducing register pressure incurred by thread divergences rather than the divergences themselves, some [8] tackle divergences directly but rely on special hardware support.

In this paper, we propose a pure software solution via runtime thread-data remapping. The basic scheme is simple: to switch the data sets that the GPU threads work on so that all the threads in a warp would take the same path on a conditional branch. Consider the "if" statement in the example mentioned earlier in Figure 1. Suppose the other (say 992) elements of $r1$ all have the similar value pattern as the first 32 elements have—that is, in every 32 elements, only one value is odd—we can remove all the thread divergences by remapping the threads to data so that the first 992 threads all work on the data with small $r1$ values, and only the final 32 threads work on the large ones. This strategy, apparently, works with the "for" loop as well in a similar manner.

A series of issues must be solved before thread-data remapping can be feasibly applied to real GPU applications. The first is the determination of a desirable thread-data mapping. In a real GPU application, data accesses may be irregular or have complex indexing expressions. The thread-data remapping may cause side effects—such as, altering original regular memory reference patterns.

The second issue is on what mechanism to use to realize the thread-data remapping. There are two options. One is through *reference redirection* (also called indirect accessing). For the "if" example in Figure 1, we may create an index array $I[\ ]$ and change $r1[tid]$ to $r1[I[tid]]$. Appropriately setting the values in $I[\,]$ will produce a desired thread-data mapping. The second option is through *data layout transformation* (also called data packing). For the "if" example again, if we keep the kernel unchanged but relocate the elements in $r1$ so that small values are all at the front of $r1$ and large ones all at the end, we can achieve the same mapping as the redirection array produces. It is necessary to explore both options to determine their limitations, effectiveness, and how they should be applied safely.

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

In this work, we develop a set of techniques to address these issues, making run-time thread-data remapping feasible for GPU computing. Our description starts with an abstract form of GPU applications and the concept of thread-data remapping (Section 2). In Section 3, we discuss the two mechanisms, reference redirection and data layout transformation, for the realization of thread-data remapping, with their properties, constraints, and suitable scenarios.

In Section 4, we present two techniques that make runtime remapping possible for GPU computing by effectively hiding and reducing remapping overhead. The first technique is a CPU-GPU pipelining scheme, which allows the remapping-related operations to overlap with GPU kernels execution. Its effectiveness in hiding remapping overhead comes from its exploitation of the massive data-parallelism in GPU applications and the independence between CPU and GPU memory systems. The second technique is a linear-time LAM (label-assign-move) scheme, which minimizes the number of data movements required for the generation of a target thread-data mapping. The two techniques can virtually remove all remapping overhead for most applications and make costly runtime remapping affordable.

Section 5 reports up to 1.47 speedup on a set of GPU applications, demonstrating the effectiveness of the techniques for eliminating thread divergences and streamlining GPU computing on the fly.

2. GPU THREAD-DATA REMAPPING

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

2.1 An Abstract Form of GPU Kernels

A GPU application (written in CUDA) consists of some CPU code and GPU code. The GPU part includes one or more functions; each of them is called *a GPU kernel*. All applications start from the "main" function in the CPU code. When the CPU launches a GPU kernel, a number of GPU threads are created with each having a unique sequential number, called thread ID and denoted as *tid* in this paper[2] Upon its creation, every thread starts an instance of the GPU kernel independently. Although they execute the same kernel with the same parameter values, they may behave differently and access different data. The differences usually stem from the uses of *tid* inside the GPU kernel.

Figure 2 outlines an abstract form for GPU kernels containing condition statements. We elide the parts irrelevant to thread divergence. We use arrays to represent container objects in GPU kernels as they are the most commonly used data structures. The *input arrays* are those arrays whose values are passed from CPU to GPU at the invocation of the

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

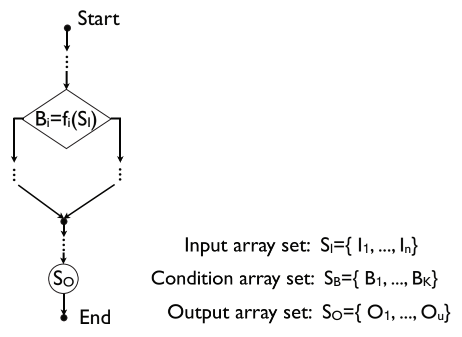

Start
$B_i=f_i(S_I)$
S_O
End
Input array set: S_I={ I_1, ..., I_n}
Condition array set: S_B={ B_1, ..., B_K}
Output array set: S_O={ O_1, ..., O_u}

Figure 2: An abstract form of a GPU kernel containing conditional statements.

kernel. Note, in this abstract form, all thread IDs together are viewed as a special input array as $IDArray[tid] = tid$. This view is important for the applicability of thread-data remapping (as Section 3.2 will show).

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

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

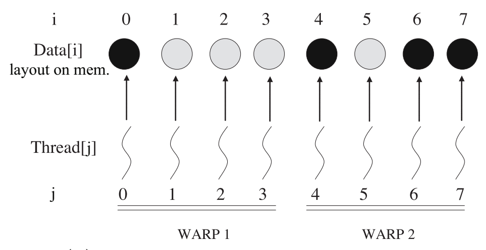
$$S_B[tid] =< B_1[tid], B_2[tid], \cdots, B_K[tid] > .$$

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

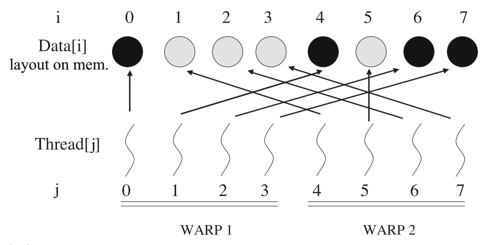
2.2 Concept of Thread-Data Remapping

It is important to note that the input set of a thread determines its behavior in a given kernel and thus the path vector of that thread. The implication is, if after a remapping, the *i*th thread maps to the original $S_I[j]$ $(i \neq j)$, then the new value of $S_B[i]$ would be the same as the original value of $S_B[j]$. This is the basic rationale for using thread-data remapping to remove thread divergences. If we view the values of the path vector produced by an input set as the color of that input set, there is no divergence if and only if all threads in a warp are mapped to the input sets of the same color. The purpose of thread-data remapping is essentially to find an appropriate mapping between threads and input sets, as Figure 3 illustrates (the two approaches to remapping will be presented in Section 3).

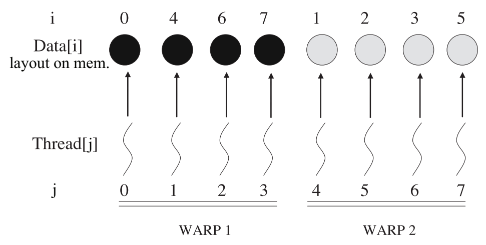
Thread divergences may come from two kinds of sources. The first is the differences in loop trip-counts (i.e. numbers of iterations), as illustrated by the "for" loop in Figure 1. Clearly, the time for a thread warp to finish the execution of the loop is determined by the largest trip-counts of the loop in the executions by all the threads in that warp. Each iteration of a loop typically takes a similar amount of time to run. Hence, the total time that the loop costs all the



(a) Original thread-data mapping



(b) Remapping through reference redirection



(c) Remapping through data layout transformation

Figure 3: Illustration of thread-data remapping for elimination of thread divergences. The two types of circles represent two types of input data sets. Their differences make threads in a warp diverge on a condition statement in (a). All divergences disappear after either of the remappings in (b) and (c).

warps is determined by

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

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

Thread divergence may also come from non-loop condition statements, as illustrated by the "if" example in Figure 1. In this case, the kernel contains no diverging loops but K $(K > 0)$ other types of condition statements. Each thread hence has a K-dimensional path vector. From now on, we say two threads are of the same type if their path vectors are equal. The basic remapping strategy in this case is to greedily pack threads of the same type together.

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

3. MECHANISMS TO REALIZE THREAD-DATA REMAPPING

There are two ways to realize a thread-data remapping: reference redirection and data layout transformation. Although they have both been studied in many CPU program optimizations (e.g., [4, 6]), their applications in GPUs have some new complexities. This section first describes the implementation and applicability of the two approaches, and then discusses their respective strengths and weaknesses.

3.1 Reference Redirection

Reference redirection creates an index array, denoted as $D[\ ]$, to generate a new thread-data mapping. If a new mapping requires thread i map to the original input data set $S_I[j]$, then $D[i] = j$. This creation step is put into the CPU code before the invocation of the GPU kernel (actually in a pipelining fashion as detailed in Section 4). The index array D is passed to the GPU kernel at its invocation.

The transformation to the GPU kernel is simple. At the beginning of the kernel, a statement like "__newtid = D[tid]" is inserted, where *__newtid* is a new local variable[3]. The transformation then replaces all occurrences of *tid* in the kernel with *__newtid*.

The transformation is applicable to kernels that contain no dependencies across threads. It can be proved that in this case, the transformed program produces the same output as the original program does. The intuition of the proof is simple: After the transformation, the computation and data accesses in all kernel instances remain the same as before, even though those kernel instances are executed by different threads. It is not hard to see that the transformation can also apply to the cases where a kernel consists of multiple sections separated by barriers and inter-thread dependencies exist only across sections but not within a section [4].

3.2 Data Layout Transformation

The second mechanism is data layout transformation. If the new mapping requires thread i map to the original input set $S_I[j]$, in this transformation, the content of $S_I[j]$ is copied to the corresponding locations in $S_I[i]$ before the invocation of the GPU kernel (again in a pipelining fashion, shown in Section 4). For instance, suppose in the original program, threads i, j, k map to $I_1[i]$, $I_1[j]$, and $I_1[k]$ respectively, but the new mapping requires these threads process $I_1[j]$, $I_1[k]$, $I_1[i]$ respectively. The transformation creates a new array I'_1 with $I'_1[i] = I_1[j]$, $I'_1[j] = I_1[k]$, and $I'_1[k] = I_1[i]$, and then replaces I_1 with I'_1 at the invocation of the GPU kernel. After the invocation of the GPU kernel, a restoration step is sometimes necessary, in which, the elements in the output arrays are reordered so that they are consistent with what the original program produces.

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

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

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

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

Theoretically speaking, there is another condition to meet: The only effect of the thread ID, *tid*, on the kernel execution is on deciding which data elements of the input and output arrays a thread references. The thread ID *tid* must not involve directly in value calculation. For example,

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

is allowed, but the following one is not:

$$O_1[tid] = I_1[tid*2] + tid \quad .$$

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

Fortunately, this third condition is easy to meet through a preprocessing step. In that step, an assistant array $IDArray$ is created before the GPU kernel invocation; its elements are set as $IDArray[tid] = newtid$. The array is then passed to the GPU kernel at the kernel's invocation as an extra input array. Inside the kernel, all references to this type of *tid* are replaced with $IDArray[tid]$. This transformation is demonstrated in the benchmark named *reduction* in Section 5.

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

3.3 Selections of the Mechanisms

The two mechanisms have their respective strengths and weaknesses. The applicability of data layout transformation is subject to some conditions as described in the previous section. In addition, the transformation usually causes larger transformation overhead than the alternative because of the data movements and restoration it requires (although this problem can be alleviated as to be shown in Section 4).

On the other hand, data layout transformation maintains certain memory reference patterns of the original GPU kernel, whereas, reference redirection does not always do so. In GPU, memory reference patterns strongly affect the effective memory bandwidth. For instance, in NVIDIA G200, if the words accessed by a warp fall into n different segments of global memory (a segment contains 32, 64, or 128 consecutive bytes), the GPU needs to conduct n memory transactions for those accesses. (When the threads in a warp access memory locations in a small range, all the references

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

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

by that warp may take only one transaction; such references are termed *coalesced memory references.*) So the changes that reference redirection causes to memory reference patterns may result in significant increases in the number of memory transactions for a kernel, and hence offset the benefits brought by the reduction of thread divergences.

In our implementation, the principle for the selection of these two mechanisms is as follows. If the reference redirection may be proved to hurt no memory reference efficiency, it is selected. Such cases may happen in two scenarios. One is that the kernel uses texture memory rather than global memory for data references: Texture memory tolerates memory pattern changes better than global memory for its use of cache. The other is that the redirection applies to calculations but not data references in the kernel. In other situations, the alternative mechanism, data layout transformation, is selected if it is applicable. Section 5 demonstrates the selection on different benchmarks.

4. TRANSFORMATION ON THE FLY

Because the values that a condition statement produces typically depend on the input data sets of the GPU application, thread-data remapping often needs to be applied during run time. However, the operations involved in the remapping are expensive. Without a careful design, the overhead may easily outweigh the benefits brought by the reduction of thread divergences. This section describes how we enable efficient runtime thread-data remapping through the use of CPU-GPU pipelining and a linear-time LAM scheme to hide and minimize remapping overhead.

4.1 Hiding Overhead through Pipelining

The first technique, CPU-GPU pipelining, tries to hide the transformation overhead by overlapping it with the execution of GPU kernels.

Figure 4 illustrates how the basic pipelining scheme can be implemented for an example GPU program. The CPU part of the original program is outlined in normal font in Figure 4 (a). It includes a loop, each iteration of which invokes an instance of a GPU kernel function *gpuKernel* to make the GPU process one chunk of the data. The bold-font line shows the inserted code to enable the pipelined thread-data remapping. As shown, a CPU thread is created before the invocation of the GPU kernel. That thread runs the function *remap()*, which contains the transformations for the remapping, including the computation of the desired mappings, the creation of indexing arrays, or the generation of the new data layout on the memory in the host machine (rather than GPU).

This pipelining scheme resembles the software prefetching in traditional CPU optimizations. The invocation of *remap()* in iteration i transforms the data to be used by the GPU kernel in the $i + \Delta$ iteration ($\Delta > 0$). The data transformations and the GPU computation hence overlap as shown in Figure 4 (b). In the first Δ iterations of the loop, the GPU kernel operates on the data that are not remapped. Those iterations constitute the warm-up stage of the pipeline. When iteration Δ or any subsequent iteration starts, the data to be processed by the GPU kernel in that iteration are already transformed and comply to the desired thread-data mapping. The remapping benefits start to show up in the GPU executions.

The actual implementation of the pipelining is more sophisticated than illustrated. We develop a flexible interface encapsulated in a *cpuThreadControl* component. Remapping functions are registered early in an execution. Transformation threads come from a thread pool, and are reused rather than created in each iteration. The implementation allows early termination of the transformation when it appears to be unprofitable (e.g., when the corresponding GPU kernel finishes). It uses asynchronous memory copy for data transfers between the host and GPU for efficiency.

A typical scenario where the pipelining scheme applies has two features. First, the invocation of the GPU kernel is within a loop as the one shown in Figure 4 so that the data are processed chunk by chunk. Second, the data chunks processed in different iterations have no overlap. Especially, the data chunks to be processed in future iterations do not depend on the previous iterations. Note that it is allowed to have some data shared by multiple iterations, as long as those data are not what the transformation procedure operates on.

Such scenarios often exist or can be created in a GPU application because of its data-parallel property. A common pattern in many GPU programs is that each thread works on a separate set of data. Inter-thread communications exist only within a block of threads; no global synchronizations are allowed in GPU kernels. If there is no such loop in the original GPU program, it is possible to partition working data into chunks for a new loop to iterate on.

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

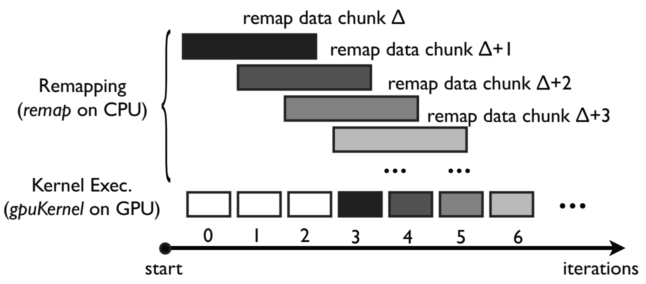
The value of Δ in the pipelining scheme decides the time distance between the remapping and the use of a chunk of data. We call it the *pipeline depth*. The suitable pipeline depth depends on the ratio (denoted as r) between the time a remapping requires and the time an execution of the GPU kernel takes. If $\Delta < r$, the transformation of a data chunk cannot finish when the GPU kernel needs to reference those data. On the other hand, if Δ is much larger than r, the warm-up stage is unnecessarily long. The appropriate value of Δ may be selected through profiling runs or runtime adaptation. For runtime adaptation, at the beginning of an execution of the GPU application, the depth may be set to 1 by default. During the execution, if the layout transformation cannot finish in time in an iteration, the depth increases by 1 to enlarge the chance for future transformations to succeed. Detailed explorations are beyond the scope of this paper.

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

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

```
.. ...
pData = & data;
for (i=0; i<N/S; i++){
   createThread (&remap, pData+Δ*S]);
   ... ...
   gpuKernel <<<... ...>>> (pData, ...);
   ... ...
   pData += S;
}
```

(a) The CPU part of an example CUDA program, with the code (in bold font) for launching the thread-data remapping process.



(b) Pipeline when the depth Δ is 3

Figure 4: An example illustrating the (simplified) use of CPU-GPU pipelining to hide the overhead in thread-data remapping transformations.

4.2 Overhead Minimization through LAM

Even though the CPU-GPU pipelining scheme helps overlap the remapping process with computation, it is still important to minimize the overhead of the remapping transformations. Large overhead leads to a deeper pipeline, which in turn causes two consequences. First, the warm-up stage of the pipeline is long, leaving many initial iterations of a loop unoptimized. Second, many transformation threads must run concurrently, resulting in a large burden to the host system. Further, increasing the pipeline depth cannot always hide the entire transformation overhead. When the number of transformation threads is so large that the capacity of CPU or memory bus is saturated, increasing the pipeline depth can only prolong the transformation time.

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

High-Level Design.

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

This scheme is named LAM (label-assign-move). Figure 5 outlines the algorithm. For easy explanation, the following description assumes that the kernel contains only one condition statement. It is a loop, which has $data[tid]$ iterations in the execution of thread tid. At the end of this section, we discuss the algorithm in a general setting.

The high-level design of LAM is that it partitions the possible values in $data$ into a number of classes, and then constructs a data layout so that most warp segments[5] are pure—that is, containing elements of the same class. To avoid unnecessary data movements, the scheme first uses a class ID to label each warp segment of the original $data$ array. The elements in a warp segment that belong to its labeled class won't be moved during the construction of the new data layout. By ignoring the differences within a class and avoiding unnecessary movements, LAM may save significant transformation overhead.

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

Detailed Algorithm.

Figure 5 outlines the algorithm of LAM. LAM includes three steps as suggested by its name. In the "label" step, it partitions the value range of array $data$ to R sub-ranges, represented by r_i $(i = 1, 2, \cdots, R)$. Each element in $data$ must belong to one sub-range (we say that the sub-range covers that element). There are two attributes associated with a sub-range, stored in $r_i.quota$ and $r_i.toFill$. Let n_i represent the number of elements in $data$ covered by r_i. The algorithm sets $r_i.quota$ to $\lfloor n_i/warpSize \rfloor$ initially, indicating the largest number of warps that can be entirely covered by r_i. The "label" step examines each warp segment in $data$. Let r_{max} be the sub-range with the largest coverage for a warp segment; the algorithm labels that warp segment with seg_{max}. Suppose an element in that warp segment a is not covered by r_i, the algorithm puts the location of a into the array $r_i.toFill$, indicating that this location should be filled with some other element of $data$ that is covered by r_i. An exception occurs when the quota of r_i is used up, when that warp segment is labeled as "mixed".

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

When the "label" step finishes, the value of *destLoc*[*i*] is the desired destination for *data*[*i*] if *destLoc*[*i*] is not zero. Otherwise, *data*[*i*] needs no movements. The final step, "move", simply copies the elements of *data* to *dataCopy*.

Discussions.

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

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

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

A key parameter in LAM is R. The larger it is, the better the resulting mapping is, in terms of eliminated divergences. But meanwhile, more overhead will be incurred by the transformation. In our experiments, we set it to 10. A desirable scheme is to dynamically determine its appropriate value through runtime adaptation, which may happen cooperatively with the adaptation of CPU-GPU pipeline depths (Section 4.1). The detail is left for future explorations.

5. EVALUATION

For the techniques to be useful in practice, we must make sure there is not only program performance improvement in divergent scenarios, but also no performance degradations in other cases. Our evaluation focuses on both aspects:

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

5.1 Methodology

Table 1 shows the five benchmarks selected for evaluating the techniques in both aspects. The first two come from real world applications, and the other three come from the NVIDIA CUDA SDK 2.0 [1]. These benchmarks contain different amount of thread divergence and hence different potential gain that can be produced by the transformation techniques. The first four benchmarks have thread divergences, suitable for the assessment of the effectiveness in thread-divergence removal. The last benchmark, which contains condition control flows, has no thread divergences. We include it to test whether the proposed techniques can ensure the basic efficiency of the program in the extreme case.

One of the difficulties we come across in the evaluation process is the lack of standard GPU benchmarks. Many previous studies have used only kernels in industry released SDKs (e.g. NVIDIA CUDA SDK). However, because those kernels are created partially to show the appealing power of GPU, most of them have virtually no GPU-unfriendly features—such as thread divergence [6]. Even for programmers of real GPU applications, as they are informed that thread divergence could be a GPU performance killer, they typically avoid using GPU if the program includes complex control flows. The consequence is the sparsity of interesting applications for the evaluation of thread divergence removal techniques, even in real application suites. The implication of such a phenomenon is not that thread divergence elimination is unimportant, but the opposite: The current exploitation of GPU is evidentially limited by the weakness in handling thread divergences; resolving such a issue may well extend the use of GPUs in high performance computing.

Table 1: Benchmarks

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

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

Our experimental platform is an NVIDIA Tesla 1060 hosted in a dual-socket quad-core Intel Xeon E5540 machine. The Tesla 1060 includes a single chip with 240 cores, organized in 30 Streaming multiprocessors. We use CUDA as the programming model.

In our experiments, the transformations are conducted in a semi-automatic manner. We implement a runtime library that includes a set of functions that facilitate both the runtime transformation and overhead reduction. These functions include the LAM scheme, the pipelining threads controller, the functions that search for suitable thread-data mappings, and so on. For each program, we manually insert invocations of the remapping in appropriate locations to enable the CPU-GPU pipelining. As the focus of this work is on the examination of the effectiveness and feasibility of the transformation techniques, we leave their integration in compilers as future work.

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

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

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

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

for (warp=0; warp < N/WZ; warp++){
   i = warp*WZ;
   l = argmax (coverage(r[k], data[i:i+WZ-1]));
         k

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

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

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

Figure 5: The LAM (label-assign-move) scheme for reducing the overhead in data layout transformation.

5.2 3D-LBM

The program, 3D-LBM, is a PDE (partial differential equation) solver based on the LBM (lattice Boltzmann model), implemented by Zhao [18] for GPU. The LBM is a model initially designed to solve fluid dynamics through the construction of simplified microscopic kinetic models. Its extended version may help solve the parabolic diffusion equation, a critical component in the elliptic Laplace and Poisson equations, widely used in the manipulation of images, surfaces and volumetric data sets. The developed LBM program is for 3D fluid simulation. The code has been highly optimized, appearing to be an efficient alternative to traditional implicit iterative solvers on CPU [18].

Our experiment focuses on a kernel named "streamAndCollision_k", one of the most time-consuming kernels. In that kernel, each thread works on a fluid node. It checks 19 directions in a 3D space, and performs density calculation and node update if it finds neighboring nodes in the a direction. As shown in Table 2, this condition statement causes 50–100% thread warps to diverge, depending on the thread block dimension and block size.

As the neighboring nodes are input-specific and compile-time unknown, the divergences can only be handled through runtime transformations. In our experiment, we use data layout transformation as the main approach to remapping because the alternative approach, reference redirection, changes memory reference patterns and hurts memory coalescing. But there are some data in the program whose layout changes are not feasible; we use reference redirection for them.

Table 2 reports the experimental results. Because the program performance is sensitive to the size of a thread block, we experiment with three different block sizes, ranging from 8 to 32 (the largest size is 32 because of the limits on the 3D space). Half to all of the 1024 warps diverge on the condition statement. The transformation removes all divergences, making the kernel run 1.17 to 1.44 times faster than the original version does. This one time data layout transformation overhead is 1900μs, smaller than the execution time of the kernel. As this is a fluid dynamic simulation application, the kernel is invoked periodically for many times; the pipelining scheme well hides all the transformation overhead.

Table 2: Experimental results of 3D-LBM

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

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

5.3 GAFORT

GAFORT is a CUDA program implemented based on an OpenMP version in SPEC OMP3.2. It computes the global maximum fitness in a genetic algorithm. The program starts with an initial population and then generates children who go through possible crossover, jump mutation, and creep mutation with given probabilities. The major computation components are implemented as GPU kernels to minimize host-device data transportation. The data is organized in order to facilitate memory coalescing. In each iteration, runtime generated random numbers determine where and how crossover, jump mutation or mutation needs to be done. The random numbers hence introduce misalignments of control

Table 3: Experimental results of GAFORT

Block size		32	64	128	256
Diverg. ratio	org	100%	100%	100%	100%
	opt	56%	56%	56%	56%
Exe. time (μs)	org	67225	67237	67243	67232
	opt	51309	51325	51377	51344
Speedup		1.31	1.31	1.31	1.31

Table 4: Experiment results of MarchingCubes

Block size		32	64	256	512
Diverg. ratio	org	100%	100%	100%	100%
	opt	0.48%	0.48%	0.48%	0.48%
Exe. time (mus)	org	17414	16707	16673	17444
	opt	12666	12371	12425	13049
Speedup		1.37	1.35	1.34	1.34

flow into the GPU kernels. The random number generator runs on CPU, interleaved with the GPU kernels.

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

The kernel contains two types of divergences. The first is due to the decision on whether a crossover is to be performed on the children genes. The second happens in the crossover computation. That computation contains a loop, whose trip-count equals the length of the segment of parent genes the child needs to use, which differs across children genes. We handle the two types of divergences by combining them into one: The case of no crossover operations is equivalent to the case the trip-count of the crossover loop as 0.

Similar to *3D-LBM*, the GPU kernel of this program uses coalescing global memory accesses for most arrays and meets the conditions listed in Section 3.3. Therefore data layout transformation is selected. A direct application of the data layout transformation would add 8000μs overhead, largely offsetting the benefits it can bring, not to mention the data restoration step we have to perform after the kernel is finished (due to the write accesses in the kernel). With the LAM algorithm and CPU-GPU pipelining, the overhead can be completely hided.

Table 3 reports the results. Unlike *3D-LBM*, this program appears to be insensitive to the thread block size. In all the cases, the transformation reduces the divergence ratio from 100% to the minimum, 56%. The minimum divergence ratio is still significant because of the large variations in the trip-counts of the crossover loop. In addition to the reduction of thread divergences, the transformation on this kernel reduces the distance between array elements accessed by adjacent threads, which increases the coalesced global memory accesses.

5.4 MarchingCubes

The Marching Cube Isosurface application is from CUDA SDK 2.0. It extracts a geometric isosurface from a volume dataset using a marching cubes algorithm. The program uses a mathematical function to create a 3D grid, and then loads the grid into the GPU device memory. After calculating the isosurface triangles, it renders the graph immediately. This program provides an interactive GUI interface, by which users can perform a variety of actions, such as rotation, zoom in, and recomputation of the isosurface. The program invokes particular kernels according to the user's input. One of the frequently used kernels is the triangle generation kernel. It is a kernel for graphics rendering. The kernel runs on the major data structure of voxels. Thread divergences occur because of the non-uniform distribution of the number of vertices which intersect the isosurface.

There are two versions of a kernel named *generateTriangle2* in the CUDA SDK. One skips the non-occupied voxels that produce zero intersected vertices, the other scans all the voxels in the computation. The first version embodies an algorithmic effort for thread divergence elimination, but because it classifies voxels only into two classes based on the number of vertices, some divergences still remain in the program. We apply our optimizations to both kernels and observe speedups ranging from 1.1 to 1.37. The following description concentrates on the second version of the kernel.

Different from the previous two benchmarks, this kernel uses texture memory. As mentioned in Section 3.3, reference redirection is an appealing option in such a scenario and is selected. The remapping takes about 120μs, less than one kernel completion time, and hence is completely hided by the CPU-GPU pipelining scheme.

Table 4 presents the experimental results. One run of *MarchingCubes* is very short; we measure and report the time of 50 runs. The speedup ranges from a factor of 1.34 to 1.37.

5.5 Reduction

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

According to the principles in Section 3.3, for this program, reference redirection is selected as the transformation technique because its influence is on the computation but not on the global memory references. The transformation follows the description in Section 3.1. It includes the creation of a new array, "newIDArray", the first half of whose elements are assigned with even values ranging from 0 to the thread block size, and odd values for the second half. In the kernel, the references to "tid" are replaced with "newIDArray[tid]" (except in the memory loading statement). After such a transformation, on the first level of the reduction tree, all of the first half of the threads in a block get involved in the calculation, hence all of the thread divergences are removed. There are still some divergences on the other levels of the reduction tree. However, as the first level covers a large portion of the kernel running time, the performance

Table 5: Experiment results of Reduction

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

improvement is evidential as shown in Table 5 on input arrays of different sizes.

We note that as shown in the NVIDIA tutorial, the thread divergences in this program can be completely removed through algorithmic changes, leading to further speedup. However, unlike the transformations conducted in our experiment, such changes require programmer's domain knowledge and complete manual efforts. And that approach is specific to the reduction problem, rather than a generally applicable, potentially automatable solution.

5.6 BlackScholes

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

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

The thread-data remapping step on the CPU computes the values of the condition variable but does not relocate any data because the condition variable values suggest no need for that. The remapping step runs in parallel with a preceding independent GPU kernel and takes more than 10 times of the kernel execution time. However, with the efficiency control, the remapping is shut down automatically; the execution time of the kernel shows no noticeable changes compared to that of the original as shown in Figure 6, indicating the effectiveness of the scheme in preserving the basic efficiency of a program.

5.7 Summary of Experimental Results

To provide a holistic view of the experimental results, we put the speedup results of all benchmarks into Figure 6.

The three bars of a benchmark in Figure 6 correspond to the normalized execution times of three versions of the kernel: the original version, the version after the thread-data remapping transformation but without efficiency control, and the version after the transformation with efficiency control. (The baseline of the normalization is the execution time of the original version.)

The large speedups of the third version demonstrate that the transformation generates significant performance improve-

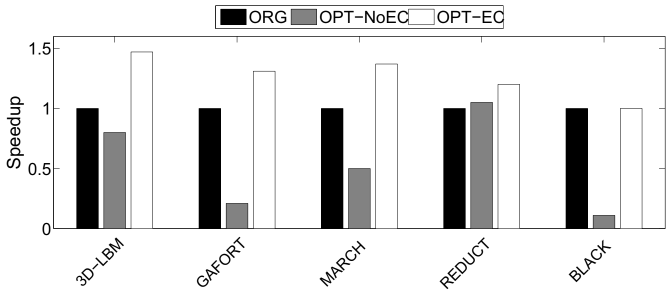


Figure 6: Speedup with and without efficiency control.

ment. The comparison with the performance of the second version reveals that the efficiency control techniques are crucial for software divergence elimination methods. The *blackscholes* results especially illustrate the importance of the efficiency control in maintaining the basic efficiency of the program in extreme cases.

6. RELATED WORK

There is a body of work focusing on the optimization of some specific GPU applications (e.g., [7, 12, 17]), building general-purpose GPU performance tuning tools [11, 14], or optimizing memory references [2, 10, 16]. Our studies are closely related to thread-divergence elimination and data transformations.

Fung and others [8] have tried to reduce thread divergences through special hardware extensions. Our techniques do not need special hardware support. Carrillo and others [3] have proposed loop splitting and branch splitting for optimizing GPU applications that contain loops and branches. Their techniques aim at the alleviation of register pressure, rather than the reduction of thread divergences. The goal of our work is complementary to theirs.

Data layout transformation has been used to reduce cache and TLB misses in CPU [4–6, 9]. We are not aware of previous uses of the transformation for thread divergence elimination in GPU. This paper explores some new challenges for data layout transformation in GPU, especially those caused by the distinctive properties of GPU architectures (e.g., memory coalescing), the conflict between the large transformation overhead and the need for runtime transformation, and the little tolerance of overhead due to the massive parallel computing power of GPUs.

7. CONCLUSIONS

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

8. ACKNOWLEDGMENTS

We owe the anonymous reviewers of ICS'10 our gratitude for their helpful comments on the paper. We thank Ye Zhao at Kent State University for providing us with the 3D-LBM benchmark. This work benefits from discussions with Xiaoming Li at the University of Delaware. The GPU devices used in this work are donations from NVIDIA. This material is based upon work supported by the National Science Foundation under Grant No. 0720499 and 0811791 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.

9. REFERENCES

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

[2] 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.

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

[4] 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.

[5] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[6] 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.

[7] Y. Dotsenko, N. K. Govindaraju, P. Sloan, C. Boyd, and J. Manferdelli. Fast scan algorithms on graphics processors. In *ICS'08: Proceedings of the 22nd Annual International Conference on Supercomputing*, pages 205–213, 2008.

[8] 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.

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

[10] 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.

[11] 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.

[12] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka. Bandwidth intensive 3-D FFT kernel for GPUs using CUDA. In *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–11, 2008.

[13] 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 '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, 2008.

[14] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, J. A. Stratton, and W. W. Hwu. Program optimization space pruning for a multithreaded GPU. In *CGO'08: Proceedings of the Sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 195–204, 2008.

[15] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens. Efficient computation of sum-products on GPUs through software-managed cache. In *Proceedings of the 22nd ACM International Conference on Supercomputing*, pages 309–318, June 2008.

[16] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W.-M. W. Hwu. Cuda-lite: Reducing gpu programming complexity. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, 2008.

[17] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.

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