One Stone Two Birds: Synchronization Relaxation and Redundancy Removal in GPU-CPU Translation

Ziyu Guo*
Qualcomm CDMA Technologies, San Diego, CA, USA
guoziyu@qualcomm.com

Bo Wu
Computer Science Department
College of William and Mary, VA, USA
bwu@cs.wm.edu

Xipeng Shen
Computer Science Department
College of William and Mary, VA, USA
xshen@cs.wm.edu

ABSTRACT

As an approach to promoting whole-system synergy on a heterogeneous computing system, compilation of fine-grained SPMD-threaded code (e.g., GPU CUDA code) for multicore CPU has drawn some recent attentions. This paper concentrates on two important sources of inefficiency that limit existing translators. The first is overly strong synchronizations; the second is thread-level partially redundant computations. In this paper, we point out that both kinds of inefficiency essentially come from a single reason: the non-uniformity among threads. Based on that observation, we present a thread-level dependence analysis, which leads to a code generator with three novel features: an instance-level instruction scheduler for synchronization relaxation, a graph pattern recognition scheme for code shape optimization, and a fine-grained analysis for thread-level partial redundancy removal. Experiments show that the unified solution is effective in resolving both inefficiencies, yielding speedup as much as a factor of 14.

Categories and Subject Descriptors

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

General Terms

Performance,Experimentation

Keywords

GPU-CPU translation, heterogeneous computing, redundancy removal, synchronization, optimization

*This work was done when Ziyu Guo was in The College of William and Mary.

1. INTRODUCTION

The quick rise of heterogeneous computing systems creates urgent demands for support of code portability between CPU and accelerators, for for avoiding device-specific code rewriting and promoting across-device cooperations.

Among many efforts along this line [2, 8, 11, 13, 19, 20], SPMD-translation has drawn some recent attentions. An SPMD-translation takes a program written in a fine-grained SPMD-threaded programming model—such as CUDA [1]—as the base code, and generates programs suitable for multicore CPUs or other types of devices. In a fine-grained SPMD-threaded program, it is typical that a large number of threads execute the same kernel function but on different data sets. Because the task per thread is usually small, parallelism among tasks are often exposed to an extreme extent. From such a form, it is relatively simple to produce code for platforms that require larger task granularities by task aggregation. SPMD-translation simplifies coding for heterogeneous devices, and meanwhile, enables seamless collaboration of different devices (e.g., CPU and GPU) as tasks can be smoothly partitioned or migrated across devices. Recent efforts in this direction have yielded translators at both the source code level (e.g., MCUDA [20,21]) and below (e.g., Ocelot [8].) A main type of target architecture of the translation is multicore CPU, in which case, the translators are also called GPU-CPU translators.

Due to the architectural differences among different types of devices, some coding tradeoff suitable for exploiting one type of processors may not fit another well. An SPMD-translation that is oblivious to such differences may end up producing code of inferior efficiency. In this work, we concentrate on two main sources of inefficiency that limits existing GPU-CPU translators.

The first source of inefficiency is in the redundant computations in GPU programs (Section 3.2.) In GPU programs, there are typically more redundant computations than in CPU programs: As GPU is strong in supporting massive parallelism but weak in handling conditional branches, it is common for a GPU program to allow some threads to carry some useless computations because otherwise, some conditional branches would have to be introduced—and these statements often hurt GPU performance substantially due to the weakness of GPU in handling conditional branches. In the parallel reduction code in CUDA SDK [1], for instance, more than half of all threads conduct some useless computations (elaborated in Section 3.2.) Such redundancies are often acceptable on GPU as they overlap with useful calculations for the massive parallelism of GPU. But when getting into the translated CPU code, they may cause serious inefficiency. In the case of parallel reduction, the useless computations in the execution of the translated CPU code weights more than half of the entire execution time. Such a redundancy differs from the traditional concept of redundancy in that they are thread-dependent. We call it *thread partial redundancy.*

The second relates with synchronizations in GPU programs (Section 3.3.) A GPU kernel is usually executed simultaneously by thousands of threads. Fine-grained synchronizations (e.g., locks) are rarely used—they are especially error-prone for large scale parallelisms. And furthermore, they often require the insertion of some thread-ID–based conditional statements for distinguishing threads with different synchronization needs. As a result, many GPU programs employ barriers, causing *overly strong synchronizations* whose constraints are stronger than necessary. In this paper, we sometimes call these synchronizations *relaxable synchronizations.*

In this paper, we propose a unified solution to both issues (Section 4.) The key insight is that the two kinds of inefficiency essentially come from a single reason: the non-uniformity among GPU threads. At a relaxable synchronization point, some but not all threads really need to synchronize; at a thread partially redundant instruction, some but not all threads really need to execute that instruction. Examining the behaviors of each individual thread is hence necessary for a GPU-CPU translation to avoid both sources of inefficiency.

Based on the insight, we develop a thread-level fine-grained analysis framework to address both types of inefficiency. Unlike existing GPU-CPU translations, the analysis target of our framework is not the static instructions but their instances executed by each thread. The framework uses *thread-level dependence graphs (TLDG)* (Section 4.1) to model the relations among the dynamic instances of GPU instructions in the executions of different threads. TLDG has a bounded size, with edges capturing critical dependences.

Based on the TLDG, we develop a CPU code generator with three novel features. The first feature is an instance-level instruction scheduler (Section 4.2), which produces a schedule for the instances of all instructions in a TLDG. The schedule is free of barriers but obeys all critical dependences. In another word, it automatically reduces the strength of overly strong synchronizations but without compromising correctness. As usually more than one legitimate schedules exist, we develop three scheduling algorithms and present a systematic comparison of their influence on the locality and performance of the generated code.

The second feature is the use of graph pattern matching and instance-level conditional branch elimination for the optimization of the generated code. The graph pattern matching works on TLDG. It automatically recognize regular patterns in the TLDG and reduces them to loops to control the size of the generated CPU code (Section 4.3). The conditional branch elimination works on instruction instances exposed by the TLDG. It precomputes some conditional statement results to reduce runtime overhead.

The third feature is a scheme for identifying thread partial redundancy from the TLDG and preventing them from getting into the generated CPU program through an integration with the code generator (Section 4.4.)

We evaluate the instance-level analysis framework on a set of GPU programs. By comparing with the codes produced by prior techniques [11, 20], we demonstrate that the techniques are able to address both types of inefficiency effectively, improving the performance of the produced CPU program by as much as a factor of 14 (Section 5.)

To the best of our knowledge, this TLDG-based framework provides the first comprehensive solution to relaxable synchronizations and thread partial redundancy problems in GPU-CPU translations. This study demonstrates the large potential of instance-level analysis, which may open many other opportunities for program optimizations in heterogeneous computing systems.

2. BACKGROUND ON GPU AND SYNCHRONIZATIONS

In this work, we concentrate on GPU programs written in CUDA due to its broad usage. This section provides some basic background on CUDA and GPU that is closely relevent to the following discussions.

Overview of CUDA. CUDA is a typical example of fine-grained SPMD-threaded programming models. It is designed for GPU, a type of massively parallel device containing hundreds of cores. CUDA is mainly based on the C and C++ languages, with several minor grammar extensions. A CUDA program is composed of two parts: the host code to run on CPU, and some kernels to run on GPU. A GPU kernel is a C function. When it is invoked, the runtime system creates thousands of GPU threads, with each executing the same kernel function. Each thread has a unique ID. The use of thread IDs in the kernel differentiates the data that different threads access and the control flow paths that they follow. The amount of work for one thread is usually small; GPU rely on massive parallelism and zero-overhead context switch to achieve its tremendous throughput.

GPU Threads and Synchronizations. When a GPU kernel is launched, the runtime usually creates thousands of GPU threads. These threads are organized hierarchically. A number of threads (32 in NVIDIA GPU) with consecutive IDs compose *a warp*, a number of warps compose *a thread block*, and all thread blocks compose *a grid.*

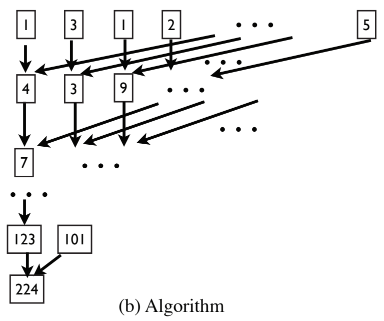
There are two main types of synchronizations on GPU. Threads in a warp run in the single instruction multiple data (SIMD) mode: No threads can proceed to the next instruction before all threads in the warp have finished the current instruction. Such a kind of synchronization is called **implicit synchronization**: No statements are needed to trigger them; they are enabled by hardware implicitly. There is another type of synchronization. By default, different warps run independently. CUDA provides a function ”__synchthreads()” for cross-warp synchronizations. That function works like a barrier, but only at the level of a thread block. In another word, no thread in a block can pass the barrier unless all threads in that block has reached that barrier. Such synchronizations are called **explicit synchronizations**.

3. SOURCES OF INEFFICIENCY

Due to architectural differences between GPU and CPU, some coding tradeoff suitable for exploiting GPU features

```
// s[ ]: (volatile) input array
for (i=blockSize/2; i>32; i>>=1){
    if (tid < i)    s[tid]+=s[tid+i];
    __syncthreads();
}
if (tid < 32){
   s[tid] += s[tid+32];
   s[tid] += s[tid+16];
   s[tid] += s[tid+8];
   s[tid] += s[tid+4];
   s[tid] += s[tid+2];
   s[tid] += s[tid+1];
}
```

(a) GPU kernel function



(b) Algorithm

Figure 1: Parallel reduction with explicit and implicit synchronizations. (warp size=32.)

may not fit CPU well. Without careful treatments, such tradeoffs, after being translated into CPU code, may cause substantial inefficiency on CPU. In this section, we examine two main sources of the inefficiency.

3.1 An Example: Parallel Reduction

To make explanation concrete, we will use the parallel reduction in the CUDA SDK [1] as an example throughout the following discussions. Parallel reduction represents a class of important operations that reduce many numbers into one or a few.

The example code from the CUDA SDK computes the sum of an input array. The execution of a thread block computes the sum of a chunk of data in the input array. The algorithm is the classic tree-shaped algorithm, as shown in Figure 1 (b). Every level of the tree computes a part of the sum.

Figure 1 (a) contains a piece of code from the GPU kernel of the reduction program in the CUDA SDK [1]. Each iteration of the "for" loop corresponds to the reduction at one level of the tree. Due to the dependences between levels, an explicit synchronization is invoked at the end of each iteration (the "__syncthreads()" call at the bottom of the loop body.)

The six lines of code at the bottom of Figure 1 (a) are for the bottom six levels of reduction. Note that there are no explicit synchronizations among the six lines, even though the same types of dependences exist among those levels as among the upper levels. This missing of synchronizations is not a coding error: Only the first warp executes these instructions and there are already implicit warp-level barriers between every two instructions. Inserting explicit synchronizations would only introduce unnecessary overhead.

This example illustrates two typical features that may cause existing GPU-CPU translators to produce inefficient code. We explain them as follows.

3.2 Thread Partial Redundancy

Thread partial redundancy refers to the case where the instances of an instruction executed by some but not all threads are useful. For instance, even though all threads in the first warp executes the bottom line of code in Figure 1 (a), only the result of the first thread's execution is useful. The redundancy is even more prominent in the reduction loop in Figure 1 (a): In all iterations except the first one, no more than half of all the threads conduct useful calculations. These redundancies are tolerable on GPU given its massive parallelism. In fact, to remove them, some thread-ID–based conditional statements have to be inserted, lengthening the critical path. However, if these useless calculations get into the generated CPU code, they may consume significant portion of CPU time, hurting the overall computing efficiency.

Thread partial redundancy differs from the concept of redundancy in the traditional compiler terminology: The instruction itself is not redundant as some threads need it to conduct some useful work. It resembles the partial redundancy in traditional compiler terms, but the distinction between being redundant or useful is not due to variances in execution paths, but due to the identity of GPU threads. As traditional partial redundant removal techniques (e.g., Lazy Code Motion [7]) are all designed to exploit execution path variations, they are not directly applicable to solve this problem.

Most existing SPMD-translations are oblivious to thread partial redundancies. Such redundancies simply become part of the generated CPU code. It is important to notice that because the instructions are not redundant (just some of their instances are), current CPU compilers do not consider them redundancy, as confirmed by our experiments in Section 5. As a consequence, these thread partial redundancies slow down the CPU executions by as much as a factor of 14 as Section 5 will show.

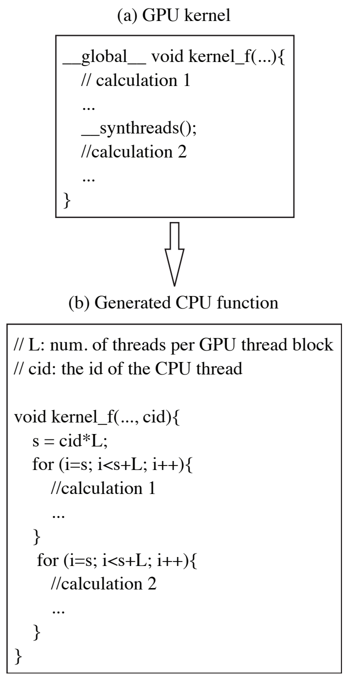
3.3 Relaxable Synchronizations

The second source of inefficiency, *relaxable synchronizations*, refers to the use of overly strong synchronizations in GPU programs. The two types of main synchronization schemes described in Section 2 are both barriers: one at the thread block level (explicit synchronizations), the other at the warp level (implicit synchronizations.) But in many cases, synchronizations are only needed between some rather than all threads in a block or warp. At the third level (i.e., the second iteration of the reduction loop) of the reduction tree in Figure 1 (b), for instance, the first thread only needs two numbers on the second level of the tree, $s[0]$ and $s[blockSize/2 - 1]$, which are computed by the first thread and the thread whose ID equals $blockSize/2 - 1$ in the previous iteration of the reduction loop. In the same vein, it is easy to see that every thread in every iteration of the parallel reduction loop may start its calculation as soon as two (or fewer) corresponding threads have finished their calculations in the previous iteration. The explicit synchronizations in the loop, as well as the implicit synchronizations between every two instructions in the bottom six lines of the code, cause unnecessarily strong constraints by requiring all threads in a thread block or warp to synchronize at those points.

The overly strong synchronizations can be relaxed with fine-grained locks. However, such locks are extremely difficult to use and error-prone at such a large scale of parallelism. Meanwhile, they would require the insertion of some conditional statements so that threads requiring different synchronizations can be distinguished from one another. These statements often hurt performance substantially due to the weakness of GPU in handling conditional branches. As a result, it is a common practice to use explicit and implicit barriers in place of fine-grained synchronizations on GPU.

3.4 Existing Treatments

Even though the thread partial redundancy and overly



```
__global__ void kernel_f(...){
    // calculation 1
    ...
    __synthreads();
    //calculation 2
    ...
}
```

```
// L: num. of threads per GPU thread block
// cid: the id of the CPU thread

void kernel_f(..., cid){
    s = cid*L;
    for (i=s; i<s+L; i++){
        //calculation 1
        ...
    }
     for (i=s; i<s+L; i++){
        //calculation 2
        ...
    }
}
```

Figure 2: Treatment to synchronizations in existing SPMD-translations.

strong synchronizations are reasonable to occur on GPU programs and are often unavoidable, they may jeopardize the efficiency of the translated CPU code when not handled carefully. In Section 5, we will see that a careful treatment to them may yield performance improvement as much as a factor of 14.

Existing SPMD-translators, however, have not given appropriate treatments to either of the two issues. They handle an explicit synchronization in a way that preserves its original semantics. For instance, MCUDA [20, 21] is a translator from CUDA kernels to C code for multicore CPU. After its translation, each code segment between two adjacent explicit synchronization points (including the start and end of a kernel) in the GPU kernel becomes a serial loop in the generated CPU code. Each of such loops has L iterations (L is the number of threads per GPU thread block), which correspond to the tasks of a thread block during the GPU kernel execution. Figure 2 illustrates this loop fission-based translation scheme. By putting the two sections of work into two loops, the translation maintains the semantics of the "__synchthreads()"—that is, no task below the synchronization point is executed before all tasks above the point have finished. Such a treatment is adopted by all other existing SPMD-translators [8, 11].

The preservation of the overly strong synchronizations in the generated CPU code has three-fold shortcomings. *First*, the loop fission limits the scope and flexibility of instruction scheduling. Merging the two loops is beyond the capability of typical compilers due to the existence of data dependences across the loops. Even though compilers and hardware may be able to schedule instructions across loop boundaries, the cross-loop instruction reordering is usually limited to the end of the earlier loop and the beginning of the later loop. *Second*, the loop fission may impair data locality. The existence of data dependences across the two loops implies data reuses between the two loops. The loop fission lengthens reuse distances and hence hurts the data temporal locality. *Finally*, the fission creates more loops and hence more loop overhead.

Implicit synchronizations in GPU code makes the efficiency problem even worse. Most prior SPMD-translations [8, 20] simply ignore implicit synchronizations. As a recent work [11] points out, such a treatment may cause violations of data dependences and hence errors in the generated CPU code. The authors propose a data dependence theorem to identify those implicit synchronizations that cannot be ignored and uses the same loop fission approach as described in the previous paragraph to tackle them. Although the approach fixes the correctness issue, it is subject to the same three efficiency issues of loop fission.

None of previous SPMD-translators has considered the thread-level partial redundancy problem. Their generated CPU code inherits these redundancy completely. As mentioned earlier, these redundancies differ from the partial redundancies current compilers handle. They may end up causing a substantial amount of useless computation on CPU.

4. A UNIFIED SOLUTION

One option to tackle the problems is to work on the generated CPU program by extending the current CPU C compilers so that it can recognize and remove the thread partial redundancy and overly strong synchronizations. Although this option may be feasible to a certain degree, it needs changes to the third-party CPU C compilers, and it has to deal with the extra complexities the code generation introduces, including the fissioned loops with partial data dependences and the interplays between the produced loop iteration space and the partial redundancy of instructions it contains.

We choose to work on the original GPU code. Besides the solution will be independent to third-party CPU compilers, there are two more reasons. First, GPU code clearly reveals the computation assignment among threads and relations among the threads, making analysis simpler. Second, the two kinds of inefficiency are essentially two forms of the non-uniformity among GPU threads: At a relaxable synchronization point, some but not all threads really need to synchronize with one another; at a thread partially redundant instruction, some but not all threads really need to execute that instruction. This key insight leads to a unified framework for solving both problems.

In the framework, we use a dependence graph at the GPU thread level (rather than statement level) to enable the examination of the behaviors of every thread and the data dependences among all dynamic instances of each GPU instruction. Such a fine-grained analysis is resilient to the non-uniformity among GPU threads, making it possible for code generator to produce CPU programs with both types of inefficiency resolved.

In the rest of this section, we will first present the thread-level dependence graph since it is the underlying vehicle for all analyses. We will then describe a set of instruction scheduling algorithms and some code shape optimization techniques for generating efficient code in the presence of synchronizations. Finally, we will discuss how thread partial redundancy can be removed easily on the TLDG framework.

4.1 Thread-Level Dependence Graphs (TLDG)

TLDG is a concept similar to traditional program dependence graphs (PDG) but with some important differences. A TLDG for a GPU kernel consists of a number of nodes and directed edges between nodes. Unlike nodes in a PDG which correspond to static statements, each node in a TLDG corresponds to a dynamic instance of an instruction in the GPU kernel. Roughly speaking, the number of TLDG nodes an instruction equals the number of threads in the scope of the TLDG. For example, for a block-level TLDG, that number is the size of a thread block. Explicit synchronization instructions have no node in a TLDG; they are treated separately. Let $node_1$ and $node_2$ represent two nodes in a TLDG, corresponding to two instructions I_1 and I_2 respectively. There is an edge from $node_1$ to $node_2$ if (1) the two nodes belong to the same thread (i.e., the same thread conducts both I_1 and I_2) and there is a control dependence from I_1 to I_2, or (2) there is a data dependence between I_1 and I_2, and I_1 may occur earlier than I_2 in an execution. Figure 3 shows an example TLDG. (For illustration purpose, the example uses source code statements rather than actual low-level instructions.)

Constructing a TLDG from a GPU kernel is straightforward. We just mention two complexities. *First*, a GPU kernel sometimes contain data races that do not hurt the correctness of executions. Upon a data race, I_1 and I_2 may happen in any order in an execution, but the edge between the two nodes can still have only one direction. The direction is arbitrarily determined by the TLDG constructor, which causes no correctness issues to the generated code because either order is allowed in the case of data races. *Second*, if there is a loop inside the kernel, it is fully unrolled before the construction of the TLDG. For a loop with large or unknown trip-counts so that it is hard to unroll, the whole loop is taken as a single node in the TLDG. Because of the regularity of data-level parallel computations on GPU, such hard-to-unroll loops are uncommon. With these operations, the resulting TLDG is directed and acyclic.

A TLDG has several properties. *First*, it exposes much more detailed dependences than traditional PDG does. In PDG, each node is a statement and there is an edge between two statements as long as some instances of the two statements have dependences. With dependences exposed for every instance and every thread, TLDG lays the foundation for tackling thread-level dependences and redundancies. *Second*, a TLDG may consist of multiple separate graphs when some threads are independent with some others. At an extreme case, when there are no cross-thread data dependences, a TLDG consists of L graphs (L equals the number of threads it models.) *Finally*, TLDG, although modeling dynamic instances, has a bounded and usually small size. Apparently, it is enough for a TLDG to contain just the nodes for a single thread block (e.g., for handling explicit synchronizations) if the dependence patterns of all blocks are identical. When some thread blocks (or warps) show different dependence patterns, a TLDG needs to be created for each class of blocks. Fortunately, thanks to the regular data-parallelism of typical GPU programs, the thread blocks of a GPU program usually follow a single pattern. The size of the TLDG of a GPU kernel is bounded by the product of the size of a thread block and the number of kinds of thread blocks in terms of dependence patterns. Due to hardware constraints, a thread block on modern GPUs contains no more than one thousand threads. A TLDG hence contains nodes and edges at the order of thousand, easy to construct and manage. When used for handling implicit synchronizations, TLDG should be created at the level of a warp. The size of such a TLDG is usually even smaller than that at the level of thread blocks.

4.2 Relaxing Synchronizations through Instance-Level Instruction Scheduling

A typical SPMD-translation result exploits parallelisms among the tasks executed by different GPU thread blocks. The computations conducted by one thread block are usually sequentially executed by a CPU thread when the translated code runs on a CPU.

The goal of instance-level instruction scheduling is to produce an order of the nodes in a TLDG so that their sequential executions produce the same results as what the executions of the corresponding GPU thread block produces. With all dependences captured by the TLDG, the code generator just needs to produce an order of the nodes such that those dependences are obeyed. *Such an order automatically relaxes the overly strong synchronizations*: No loop fission is needed any more because the successful preservation of instance-level dependences by the node order effectively eliminates the need for a whole block synchronization.

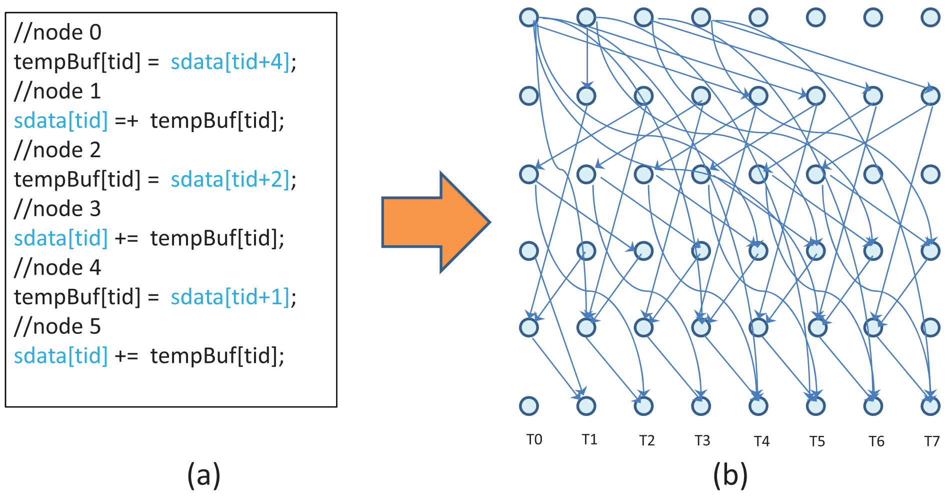
Many orders of the nodes may be legitimate. At a first glance, the scheduling problem may seem identical to traditional instruction scheduling problem except that the dependence graphs encode dependences among dynamic instances of instructions rather than static statements. Traditional list-based scheduling algorithms [7] seem to be directly applicable.

However, a careful examination shows that some unique features of our problem add some special complexities. In traditional list-based scheduling, each node in the dependence graph is labeled with some weight typically equaling the length (in terms of cycles) of the shortest path from the node to a root (i.e., a node with no outgoing edges) of the dependence graph. These weights are used as guidance for scheduling. On a TLDG of a GPU kernel, however, often hundreds of nodes have the same weights. The reason is that many threads in a block have the same dependence patterns and instructions due to the regular data-level parallel computations in most GPU kernels. The implication is that if list-based scheduling is used, the mechanism for tie-breaking will play an especially important role in the scheduling. Another important feature of TLDG is that a TLDG often consists of many separate graphs. As a consequence, some practices natural to traditional list-based scheduling will show some new effects on TLDG. For instance, if we apply the basic list-based scheduling to TLDG so that every time the node with the largest weight is taken, then the largest-weight nodes from different graphs (corresponding to different threads) will tend to be scheduled close in time, even though they may have no dependences (hence data reuses) among one another.

We design three scheduling algorithms to meet the special complexities of the TLDG scheduling problem. These algorithms are based on different heuristics, but with the same goal as to enhance data locality in the generated code.

Breadth-First Scheduling.

The first algorithm is a breadth-first scheduling algorithm.



```
//node 0
tempBuf[tid] =  sdata[tid+4];
//node 1
sdata[tid] =+ tempBuf[tid];
//node 2
tempBuf[tid] =  sdata[tid+2];
//node 3
sdata[tid] += tempBuf[tid];
//node 4
tempBuf[tid] =  sdata[tid+1];
//node 5
sdata[tid] += tempBuf[tid];
```

Figure 3: (a) The bottom three statements of Figure 1 with shared memory references broken into pieces; (b) the corresponding TLDG with only eight threads shown.

It partitions the TLDG nodes into a number of groups and imposes strict order among groups. In each round, the algorithm finds all nodes that have no incoming edges and put them into a group. It schedules the nodes in the group based on an ascending order of their GPU thread IDs. It then removes all the nodes in the group along with their outgoing edges from the TLDG and proceeds to the next round. The process continues until the TLDG becomes empty. It is easy to see that the produced code maintains the source-sink relationships of all the dependences in the original TLDG. This algorithm is called *breadth-first* because during its scheduling, nodes of different threads but at a level similar to the level of the current node are often preferred over the successors of the current node.

The breadth-first scheduling can be viewed as a variation of the basic list-based scheduling. When all threads have the same instruction sets and dependence patterns, the basic list-based scheduling algorithm, coupled with the use of thread ID as the tie-breaker, will produce the same results as the breadth-first algorithm does.

The intuition of the breadth-first algorithm is that even though the nodes in a group tend to come from different threads, they often reside at the same level in the TLDG, hence likely corresponding to the instances of the same memory access instruction in a GPU kernel. Because many GPU programs conduct coalesced memory references (i.e., adjacent threads access memory locations close to one another), a memory reference instruction in a GPU kernel often produces relatively regular memory accesses: Locations accessed by threads with adjacent IDs tend to be close in memory. Therefore, the breadth-first scheduling algorithm, by ordering nodes based on their thread IDs in each round, is likely to yield good spatial locality.

Depth-First Scheduling.

While the breadth-first algorithm may offer good spatial locality, it considers no temporal data reuses. Furthermore, it requires many temporal variables to store the intermediate computing results, and the consumption of the results is likely in another round and many instructions later.

The depth-first scheduling algorithm emphasizes intra-thread data reuses. It starts by taking a node that has the largest weight but has no incoming edges. It then removes that node and its outgoing edges from the TLDG. In each of the following steps, it tries to find a node of the same thread that the last-taken node belongs to. (The tie-breaking is thread ID.) If none of such nodes is free (i.e., having no incoming edges), the scheduler will switch to the nodes of a different thread and continue this process.

The selection of which thread to switch to is important for the performance of the produced schedule. The criterion used in the depth-first algorithm is based on the number of useful nodes in a thread. A useful node is a node containing non-redundant instructions. (The recognition of useful nodes will be described in the next section.) Because the goal of this scheduler is to divide the nodes of a thread into as few partitions as possible, the scheduler selects the thread with the fewest useful nodes left. The rationale is that by choosing such a thread, we have a better chance of removing all the nodes of that thread from the TLDG early so that the nodes in that thread will not affect the scheduling of the nodes of other threads any further.

Length-First Scheduling.

The length-first algorithm is a variation of the depth-first algorithm. Instead of limiting the selection of nodes within one thread, this algorithm prefers the selection of nodes on the longest uninterrupted path. A path is uninterrupted if all the nodes in the path have one incoming edge and that edge comes from a node in this path. The nodes in a path may belong to different threads.

Both the depth-first and length-first algorithms follow the dependence edges in the graph. The difference is that the former emphasizes intra-thread dependences, while the latter does not. As dependences means data reuses, both algorithms tend to produce code with temporal data locality highlighted.

4.3 Code Shape Optimization

With the support of the TLDG, two optimizations can be easily materialized to enhance the code shape of the generated program.

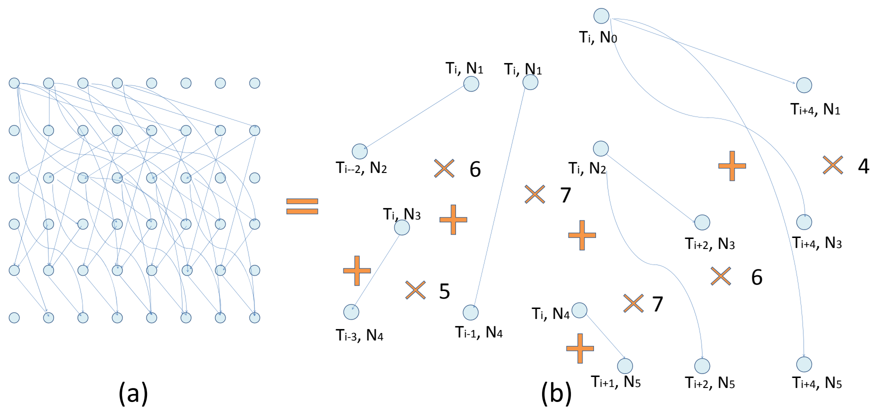


Figure 4: The original TLDG (a) broken down into six basic patterns (b), each of which retains its shape and orientation in the whole graph, only repeated in the horizontal direction.

The first optimization is enabled by the instance-level code generation. With each instance of an instruction explicitly expressed, it becomes possible to compute the results of some thread-ID–dependent conditional statements (e.g., "if (tid<32)"), and hard-code the results into the generated code. This optimization help reduce the number of instructions and branches in the generated CPU code.

The second optimization is based on the observation that a TLDG typically consists of some identical subgraphs. For example, the TLDG in Figure 4 (a) is essentially composed of the many copies of the several subgraphs in Figure 4 (b). Each of the subgraphs repeats multiple times horizontally. Such repetitions stem from the similarity of the control flows taken by the GPU threads. Such a similarity is common due to the data-level parallel computations in the kernel. The patterns can be discovered through heuristic graph pattern matching algorithms [23]. This optimization can help reduce the size of the CPU code: n copies of a subgraph can be implemented with a loop of n iterations. The application of this optimization is optional. It can be used to help strike a good tradeoff between code size and loop unrolling benefits.

4.4 Thread Partial Redundancy Removal

The TLDG provides a convenient representation for detecting thread partial redundant calculations. Because every instance of an instruction becomes a node in the TLDG, thread partial redundancy becomes complete redundancy at some nodes. The redundancy removal problem turns into the detection of useless nodes and the prevention of them from getting into the generated code.

With the support of TLDG, the detection of the redundant nodes is similar to traditional redundancy removal. The process starts with marking the initial set of useful nodes, which consists of three classes of nodes: the nodes containing returning instructions, the nodes modifying the kernel function arguments, and the nodes storing data into pinned memory (i.e., a segment of host memory directly accessible by GPU kernels.) All these nodes are very likely to contain operations useful for the program final output. The compiler then starts from these "useful" nodes and traces upwards to the top of the TLDG, marking all the nodes encountered as useful. All the unmarked nodes are then removed from the TLDG. Figure 5 illustrates this process. This optimization should happen before the instruction scheduling.

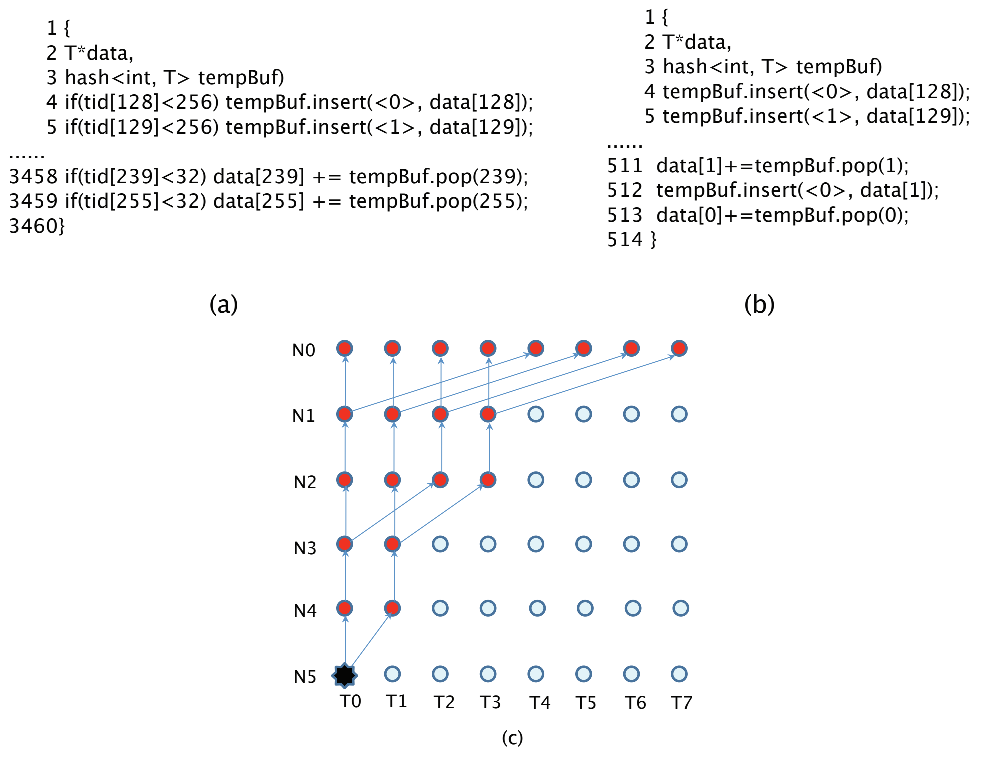
It is easy to see that both the thread partial redundancy removal and the scheduling-based relaxation of synchronizations are applicable at the levels of both thread blocks and warps. The only difference is at which level the TLDG is created.

5. EVALUATION

We evaluate the TLDG-based optimizations on five CUDA programs. Three of them come from NVIDIA CUDA SDK [1]: *Reduction* is the fifth version of parallel reduction in the SDK, *Transpose* implements parallel matrix transposing, *SortingNetworks* is an implementation of the standard sorting network model which uses a network of wires and comparator modules to sort a sequence of numbers in parallel. The program *CG* is the conjugate gradient program from NPB benchmark suite, recently ported to CUDA under the HPCGPU project. The program *Nw* implements a widely used sequencing algorithm in bioinformatics. It is part of the Rodinia benchmark suite [6]. We select these programs mainly because they form a set containing the typical GPU program features that are relevant to this work: non-trivial explicit and implicit synchronizations, and thread partial redundancy.

We compare the performance of four versions of the SPMD-translated code. Three of them are the versions from the TLDG-based methods described in this paper. They are named *breadth-first*, *depth-first*, and *length-first*, corresponding to the three scheduling algorithms presented in Section 4.2 respectively. By default, thread partial redundancy removal has been applied to all these TLDG-based methods. The baseline for our comparison is the state of the art—that is, to use the MCUDA approach [20] to do normal translation, and use statement-level dependence analysis [11] to identify potentially critical implicit dependences and treats them with loop fission. The generated version is named *splitting-oriented.* Recent explorations [11] have shown that among all existing GPU-CPU translation methods, this approach generates the most efficient code that runs correctly.

All our experiments are carried out on a quad-core Intel Xeon E5460 machine, with Linux 2.6.33 and GCC 4.1.2



```
{
T*data,
hash<int, T> tempBuf)
if(tid[128]<256) tempBuf.insert(<0>, data[128]);
if(tid[129]<256) tempBuf.insert(<1>, data[129]);
......
if(tid[239]<32) data[239] += tempBuf.pop(239);
if(tid[255]<32) data[255] += tempBuf.pop(255);
}
```

```
{
T*data,
hash<int, T> tempBuf)
tempBuf.insert(<0>, data[128]);
tempBuf.insert(<1>, data[129]);
......
data[1]+=tempBuf.pop(1);
tempBuf.insert(<0>, data[1]);
data[0]+=tempBuf.pop(0);
}
```

Figure 5: (a). The original generated code before redundancy removal. (b). Pruned code where all useless computations are removed. (c). Bottom-up redundancy removal, starting with the initial useful nodes, marked black.

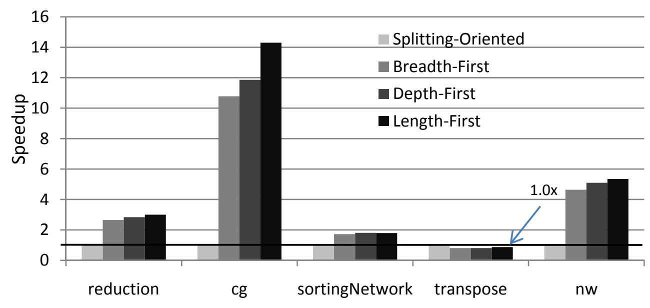


Figure 6: Performance of the benchmarks normalized by the *splitting-oriented* performance. The thread partial redundancy removal is applied.

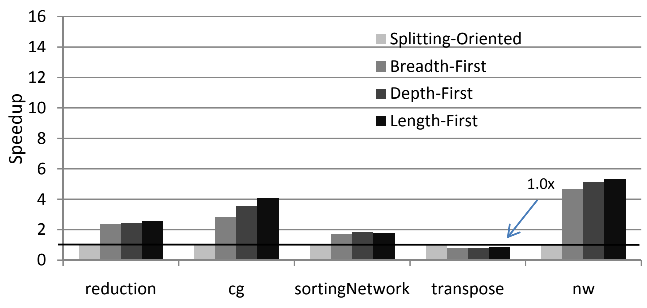


Figure 7: Performance of the benchmarks normalized by the *splitting-oriented* performance. The thread partial redundancy removal is not applied.

installed. The compilations always use the highest level of optimization.

5.1 Overall Speedup

Figure 6 reports the overall performance of the benchmarks. By breaking the thread tasks into many nodes, the TLDG-based methods do introduce additional overhead, especially in loading and storing intermediate results when two adjacent nodes from the same GPU thread are separated apart by some other nodes scheduled by the TLDG-based methods. In the case of *transpose*, with only four memory references per GPU thread and no redundant calculations, the benefits by the TLDG methods turn out to be not enough to outweigh the overhead, resulting in slightly lower performance than the code produced by the statement-level method.

However, on all the other four benchmarks, the TLDG-based methods consistently outperform the statement-level, splitting-oriented method. Since the TLDG method converts all operations in a GPU block to a piece of linear CPU code, the scope for compiler optimizations on such CPU code is much larger than in the thread loop structure generated by MCUDA-like methods. Instruction re-scheduling and merging on both compiler and processor pipeline level will have sufficient space to operate. It further reduces loop overhead. The instance-level redundancy removal proves to be a powerful tool in reducing unnecessary operations in the translated CPU code. Its effectiveness shows up in the results of *Reduc-*

tion and *CG*, both of which contains a non-trivial amount of thread partial redundant computations. For comparison, Figure 7 reports the speedups of the benchmarks without thread partial redundancy removal applied.

The breadth-first scheduler delivers the lowest performance among all TLDG-based methods. It is mainly because the scheduling algorithm is more oriented at processing the TLDG itself rather than the GPU program the TLDG stands for. Although it tries to take advantage of the largely well-arranged access patterns of the GPU kernels by sorting within each round, the benefit of that sorting turns out to be not as significant as temporal locality in the CPU code it generates. The depth-first scheduler generally performs better than the breadth-first scheduler. It is better at handling intra-thread control dependences by trying to put adjacent nodes of the same thread together in the CPU code, which eliminates the need for many temporary buffer accesses. However, it still ignores the inter-thread data dependence edges, which constitute a majority of the edges in a typical TLDG. By taking advantage of such dependence edges, the length-first scheduler achieves the highest speedup overall. The length-first scheduling generally adapts better to dependence patterns, and is therefore exposed to more temporal-reuse opportunities than other scheduling algorithms. Recall that both depth-first and length-first schedulers use GPU thread ID as the tie breaker. Because there are a large number of ties in a typical TLDG, these two methods turn out to achieve most of the spatial locality benefits that the breadth-first method obtains, even though neither of them does intra-round sorting explicitly.

From the benchmark aspect, the tested kernels of *CG* and *Reduction* are both implementation of the classic parallel reduction algorithms, but CG has more complicated computation in each iteration, and a larger block size, which translates into more data dependence edges and partial redundancy. Both factors contribute to the more prominent gains of the TLDG-based methods on *CG* than on *Reduction*. *SortingNetworks* and *Transpose* have no instance-level redundant computation, but *SortingNetworks* has a much denser graph than *Transpose*, leading to considerable benefits of the TLDG-based methods. *Nw* has few threads in a block but a large number of nodes in a thread. More than half of the dependence edges in its TLDG connect two nodes of the same thread. All three TLDG-based methods perform well on this program.

Overall, the length-first scheduling algorithm performs the best. Paired with the thread partial redundancy removal and other optimizations, it produces performance improvements from a factor of 1.9 to 14.2 on four of the five benchmarks. Although it results in a slight slowdown on one benchmark, that negative effect can be mostly avoided by simply not applying TLDG-based optimizations to small kernels—such as those with fewer than six memory operations.

5.2 Code Shape Optimization

The capability of the code shape optimization is demonstrated by applying this optimization on *SortingNetworks*, a benchmark with the densest TLDG among all the five benchmarks. As stated in earlier sections, this optimization seeks to reduce the size of the generated code by extracting those vertically non-overlapping, horizontally repetitive sub-graphs in the TLDG, and use a loop in the generated code to represent their repetitive occurrences.

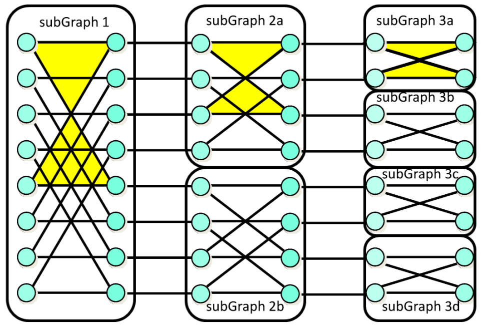


Figure 8: The repetitive subgraphs in the TLDG for *SortingNetworks*.

The optimization is applied to *SortingNetworks* TLDG, then a modified breadth-first instruction scheduler is used to generate the CPU code. The experimental result shows a two-fold reduction in the size of the code. The reason for such a substantial reduction is attributed partly to the dependence patterns in the benchmark. *SortingNetworks* employs a butterfly network approach to parallelize the comparison and swapping. In this iterative process, every entry is compared to another entry that is s entries away from itself. The variable s is reduced by half in each iteration, which essentially divides the operations in each iteration into several independent and homomorphous partitions, as shown in Figure 8 (the yellow in the figure highlights the butterfly-shape patterns.)

The reduced code shows no measurable performance difference compared to the version without code size reduction. This is due to the large instruction cache on the host machine. The benefits of code reduction are expected to be more prominent on mobile CPUs.

6. RELATED WORK

SPDM-translation has received a number of recent explorations. MCUDA [21] is one of the first studies trying to translate GPU CUDA code into C code for multicore CPU. The authors later developed a set of optimizations to enhance the code quality [20]. Ocelot [8] tries to do the translation at the CUDA PTX level. Neither of the two explorations has considered relaxable synchronizations or thread partial redundancies—the main focus of this current work.

Neither MCUDA nor Ocelot has provided appropriate treatments to implicit synchronizations in a GPU kernels. They are subject to translation errors on a program with critical implicit synchronizations.

A recent study [11] points out the correctness pitfall and proposes a dependence theorem to help identify critical implicit synchronizations—that is, those synchronizations that cannot be ignored during the translation. The authors further compare the efficiency of two code generation methods that handle critical synchronizations correctly. There are several significant differences between that study and our current work. First, all the methods proposed in the previous study are based on static statements in a GPU kernel. This current work focuses on dynamic instances of instructions. Second, this work tackles both explicit and implicit synchronizations, while the previous study has only implicit synchronizations as its target. Third, the previous study does not deal with thread partial redundancy. Finally, none of the techniques proposed in this work—including the use of TLDG, instance-level instruction scheduling, code shape optimizations, thread partial redundancy removal—is proposed in the previous study. As Section 5 has shown, these new techniques help produce code significantly more efficient than what the previous approach generates.

There is a body of work trying to simplify GPU programming. Some [3, 13] try to develop automatic compilers to translate CPU code into GPU code. They usually use pragmas to incorporate programmers knowledge. Some other work add extensions to CUDA or OpenCL (e.g. [19].) Some work exploits cooperation of CPU and GPU for whole system synergy [15, 17, 24]. Most of them assume the code for both devices are given or consider only CPU-to-GPU code translation.

There have been many studies in optimizing GPU programs for high efficiency. Some propose hardware extensions (e.g. [9, 16, 22]), but more rely on software analysis, transformations, and tuning (e.g. [4, 5, 12, 14, 18, 25–27].)

None of these studies have concentrated on the two issues this work addresses: thread partial redundancy and relaxable synchronizations. Our recent workshop paper [10] raises the two issues for the first time, but in a preliminary manner. Mainly to express our position on the issues, the workshop paper briefly discusses the properties of the two issues and the basic concept of TLDG. This current paper provides a much more comprehensive explorations to the problems in almost every aspect. (1) In the TLDG concept and construction, this paper describes how to handle hard-to-unroll loops, data races, and reveals several important properties of TLDG by comparing with traditional PDG. (2) In instruction scheduling, this paper presents three scheduling algorithms and a set of empirical comparisons among the algorithms, while only the breadth-first algorithm is mentioned in the previous workshop paper. Furthermore, this current paper systematically analyzes the difficulties for traditional list-based scheduling algorithms to apply to the instruction scheduling problem. (3) In code optimizations, the previous workshop paper contains no code shape optimizations at all. (4) Finally, this paper gives a more comprehensive evaluation of the techniques.

7. CONCLUSION

This paper has presented an exploration for enhancing the treatment of two important features of GPU kernels during SPMD-translations, namely overly strong synchronizations and thread partial redundancy. It first explains the concepts of the two features and their influence on the quality of SPMD-translation results. It then points out that the two features essentially come from a single reason: the non-uniformity among threads. It introduces the use of TLDG as a convenient representation for analyzing both issues, and proposes a set of techniques on TLDG for addressing the inefficiency caused by those features. These techniques include three instance-level instruction scheduling algorithms, some code shape optimizations, and the detection of thread partial redundancies. Together, these techniques lead to an approach which systematically relaxes overly strong synchronizations (both explicit and implicit ones) and removes thread partial redundancies for SPMD-translations. In an experiment on five programs, these techniques enhance the efficiency of the SPMD-translation results by up to a factor of 14, demonstrating the promise for improving the quality of existing SPMD-translators. The instance-level analysis proposed in this work may open many other opportunities for promoting the whole-system synergy in modern heterogeneous computing systems.

Acknowledgment

We owe the anonymous reviewers our gratitude for their helpful suggestions on the paper. This material is based upon work supported by the National Science Foundation Grant (0811791), NSF CAREER (0954015), DOE Early Career Award, 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 NSF, DOE, or IBM.

8. REFERENCES

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

[2] OpenCL. `http://www.khronos.org/opencl/`.

[3] E. Ayguade, R. M. Badia, D. Cabrera, A. Duran, M. Gonzalez, F. Igual, J. Jimenez, Daniel andL̃abarta, X. Martorell, R. Mayo, J. M. Perez, and E. S. Quintana-Ortí. A proposal to extend the openmp tasking model for heterogeneous architectures. In *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, IWOMP '09, pages 154–167, 2009.

[4] 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.

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

[6] 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.

[7] K. Cooper and L. Torczon. *Engineering a Compiler.* Morgan Kaufmann, 2003.

[8] G. Diamos, A. Kerr, and M. Kesavan. Translating gpu binaries to tiered simd architectures with ocelot. 2009.

[9] 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.

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

[11] Z. Guo, E. Zhang, and X. Shen. Correctly treating synchronizations in compiling fine-grained

spmd-threaded programs for cpu. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 2011.

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

[13] 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.

[14] 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.

[15] C. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the International Symposium on Microarchitecture*, 2009.

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

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

[18] 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.

[19] M. Steuwer, P. Kegel, and S. Gorlatch. Skelcl âĂŞ a library for portable high-level programming on multi-gpu systems. In *Proceedings of International Parallel and Distribute Processing Symposium (IPDPS)*, 2011.

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

[21] J. Stratton, S. Stone, and W. Hwu. Mcuda: An efficient implementation of cuda kernels for multi-core cpus. 2008.

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

[23] G. Valiente. *Combinatorial Pattern Matching Algorithms in Computational Biology Using Perl and R*. CRC Press, 2009.

[24] B. Wu, E. Zhang, and X. Shen. Enhancing data locality for dynamic simulations through asynchronous data transformations and adaptive control. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2011.

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

[26] E. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

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