Fine-Grained Treatment to Synchronizations in GPU-to-CPU Translation

Ziyu Guo and Xipeng Shen

College of William and Mary, Williamsburg VA 23187, USA,
{guoziyu, xshen}@cs.wm.edu

Abstract. GPU-to-CPU translation may extend Graphics Processing Units (GPU) programs executions to multi-/many-core CPUs, and hence enable cross-device task migration and promote whole-system synergy. This paper describes some of our findings in treatment to GPU synchronizations during the translation process. We show that careful dependence analysis may allow a fine-grained treatment to synchronizations and reveal redundant computation at the instruction-instance level. Based on thread-level dependence graphs, we present a method to enable such fine-grained treatment automatically. Experiments demonstrate that compared to existing translations, the new approach can yield speedup of a factor of integers.

1 Introduction

For their advantages on computing power, cost, and energy efficiency, Graphic Processing Units (GPU) have become a type of mainstream co-processors in modern computing systems. Recent years have seen a rapid adoption of GPU-specific programming models, such as NVIDIA CUDA.

GPU-to-CPU translation aims at compiling code written in GPU programming models to (multi-core) CPU code. It has drawn some recent research interest from both academy and industry [3, 8, 10, 12, 14, 16], for three reasons. First, it extends the range of applicable architecture and hence the impact of GPU programming models. An application developed in CUDA, for instance, can be automatically converted to a form suitable for multicore CPU. Such a capability is becoming increasingly important, given that the number of applications written in GPU programming models has increased continuously. Second, the translation enables smooth CPU-GPU collaborations. Given the trends towards heterogeneous systems, an essential requirement for maximizing computing efficiency is the synergistic cooperation among various types of processors. Automatic GPU-to-CPU translation facilitates seamless migration of jobs among GPU and CPU, hence helping promote a whole system synergy. Finally, a viable and widely accepted methodology for programming heterogeneous Chip Multiprocessors (CMP) systems is yet to be developed. As a programming model that exposes multi-level parallelism of an application to an extreme extent, CUDA-like GPU programming models are potential contenders if translation from them to a conventional commodity microprocessor works well.

Recent studies have produced some GPU-to-CPU translation systems, at both the source code level (e.g., MCUDA [9, 10]) and below (e.g., Ocelot [8]). However, none of them has systematically explored the different implications of synchronizations on GPU and CPU.

On GPU, it is typical that some explicit device-specific intrinsics are introduced to enable synchronizations among threads. These intrinsics are used widely for being essential for the implementation of various parallel operations on GPU. In a CUDA application, for example, the GPU threads are organized in a number of *thread blocks*, and each thread block contains a number of *thread warps*. An intrinsic, __syncthreads(), serves as a thread-block–level barrier, ensuring that none of the threads in a block passes over the synchronization point before all threads in the block reaches that point.

In many GPU applications, the synchronization intrinsics are used even though a synchronization between just a subset of threads in a block is necessary. By doing that, the programmer introduces unnecessarily strong constraints, but avoids thread-specific checks and obtains programming easiness. It causes almost no issue to the performance of GPU applications because of the low overhead and high parallelism of hardware. However, a literal translation of such __syncthreads() calls to CPU, as existing GPU-to-CPU translation systems all do, often leads to considerable inefficiency.

The problem becomes even more serious when implicit synchronizations are taken into consideration. Due to the hardware implementation of GPU, synchronizations are sometimes realized in an implicit manner. In CUDA, every thread warp (32 threads) proceeds in lockstep. In another word, none of the threads can proceed to the next instruction until all threads in the warp have finished the current instruction. This default SIMD execution model is equivalent to that there is an implicit warp-level barrier after every instruction. Due to the prevalence of such implicit synchronizations, a literal translation of GPU synchronizations to CPU would cause serious efficiency issues. Existing GPU-to-CPU translation systems typically ignore such implicit synchronizations during the translation, the consequence of which is even more serious than the efficiency issue: The produced CPU code may be erroneous for violating some data dependences originally maintained by the implicit synchronizations (illustrated in Section 3).

In a recent study [12], we analyzed the correctness issue caused by the neglication of implicit synchronizations, proposed a dependence theory to identify critical implicit synchronizations (i.e., those that should not be ignored), and developed a state-level code generation algorithm to fix the issue.

In this paper, we investigate the potential of fine-grained treatment to the synchronizations. Unlike the prior study, this exploration deals with both implicit and explicit synchronizations. More importantly, it analyzes dependences among threads at the level of the dynamic instances of GPU statements. By lowering the granularity from statement to statement instances, it achieves a more detailed understanding of inter-thread data dependences. The understanding then leads to more efficient treatments to GPU synchronizations and the revelation of fine-grained redundant computations.

In the following parts of this paper, we first discuss the origin, forms, and performance implications of GPU synchronization intrinsics, both the explicit (Section 2) and implicit (Section 3). We then present the use of thread-level dependence graphs (TLDG) for representing fine-grained data and control dependences among dynamic instances of GPU kernel instructions. We report two uses of the fine-grained analysis. The first is to generate CPU code with unnecessary synchronization constraints relaxed. The second is to prune instruction-instance–level redundant computations (Section 4). We evaluate the effectiveness of the techniques on three GPU programs. By comparing with the codes produced by prior techniques, we demonstrate that the fine-grained treatment to synchronizations has some clear performance benefits (Section 5).

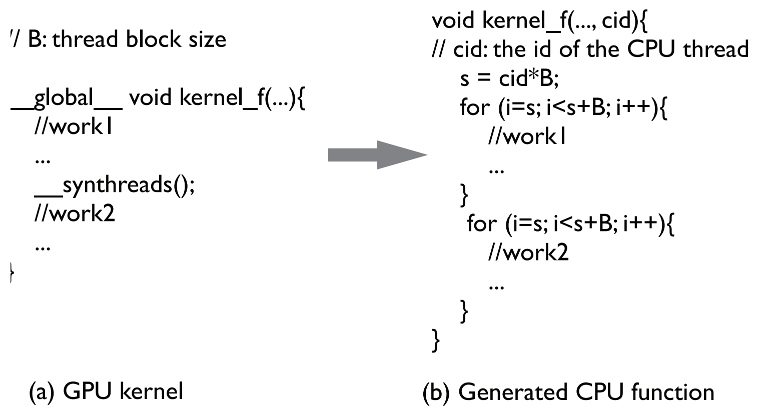
2 Impact of Explicit Synchronizations

One of the major device-specific features of the CUDA programming model is the intra-block synchronization. An invocation to the "__syncthreads()" will stall any run-ahead threads in the block until all threads have reached that point. As an essential and efficient barrier primitive, "__syncthreads()" exists prevalently in CUDA programs.

In existing GPU-to-CPU translation, the common approach to transforming GPU __syncthreads() into equivalent CPU code is to imitate the strict intra-block barrier via loop splitting. Figure 1 illustrates the translation scheme implemented in MCUDA [10], a typical GPU-to-CPU translator. For the purpose of explanation, the figure shows only the produced sequential code corresponding to the execution of the tasks done by one thread block. The GPU kernel function, orginially executed by every thread in the block, turns into two loops, which each has B iterations corresponding to the tasks executed by B GPU threads in the thread block. The two loops are for *work1* and *work2* respectively. Putting them into two separate loops ensures that the order constraints imposed by __syncthreads() are satisfied.

The loop splitting approach is subject to several drawbacks. First, it introduces extra loop overhead.

Second, it imposes strong constraints on the scheduling of instructions. In principle, the instructions in the second loop cannot be executed before the first loop finishes (unless the compiler finds that the two loops can be fused together). Although that constraint is the same as what the GPU synchronization intrinsics implies, it is often unnecessarily strong. The efficient synchronization and uniform SIMD execution model on GPU makes it attempting for programmers to skip fine-grained data dependence analysis during the coding of GPU kernels, and insert __syncthreads() wherever it might be needed. Often, synchronizations are only needed between a subset of threads. But this strategy is fine for GPU programming because the synchronization intrinsic is lightweight and usually there are no better alternatives—inserting conditional statements often intro-



```
'/ B: thread block size

__global__ void kernel_f(...){
   //work1
   ...
   __synthreads();
   //work2
   ...
}
```

```
void kernel_f(..., cid){
// cid: the id of the CPU thread
   s = cid*B;
   for (i=s; i<s+B; i++){
      //work1
      ...
   }
    for (i=s; i<s+B; i++){
      //work2
      ...
   }
}
```

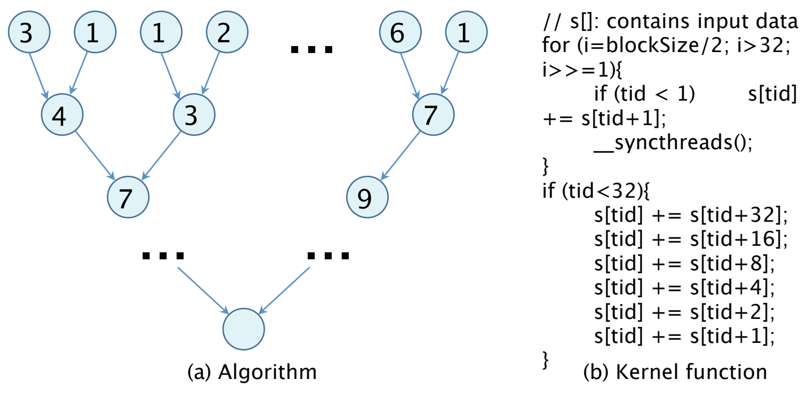
(a) GPU kernel
(b) Generated CPU function

Fig. 1. Illustration of MCUDA compilation. For illustration purpose, it shows the CPU code that corresponds to the execution of only one GPU thread block.

duces even higher overhead than synchronizations. But on CPU, the constraints may hurt instruction scheduling and hence computing efficiency substantially.

The severity of the two issues depend on the density of the invocation of synchronization intrinsics. Unfortunately, due to the simplicity brought by invocations of the intrinsics, the density can be quite high in real GPU applications. In one of our benchmarks, CG_CUDA, for instance, 23 __syncthreads() invocations appear in a kernel with only 170 lines of code.

3 Impact of Implicit Synchronizations



```
// s[]: contains input data
for (i=blockSize/2; i>32;
i>>=1){
      if (tid < 1)      s[tid]
+= s[tid+1];
      __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];
}
```

Fig. 2. Algorithm and excerpted code of CUDA SDK reduction (kernel 5).

GPU threads in a warp proceed in locksteps, which is equivalent to having an warp-level synchronization after every instruction. We call such order constraints implicit synchronizations.

Most previous GPU-to-CPU translations give no considerations to implicit synchronizations. Consequently, there is no guarantee that the generated code will retain the dependences in the original program; the produced code can be hence incorrect.

The parallel reduction program in Figure 2 illustrates the correctness pitfall. On the left is an illustration of the standard parallel reduction algorithm. It proceeds level by level; data dependences exist between every two levels. The bottom six lines of code in the kernel shown on the right of the figure correspond to the six bottom levels of the parallel reduction algorithm. They contain no explicit invocation of synchronization intrinsics, but the data dependences among the levels are well preserved, thanks to the implicit synchronizations among GPU instructions[1]. A translation of the kernel by MCUDA will violate the inter-level data dependences because of the negliction of the implicit synchronizations.

Such an exploitation of implicit synchronizations saves invocations of explicit synchronization intrinsics, and has served as a trick adopted by many GPU applications for achieving high performance. It is important to find an approach to handling them correctly and efficiently.

4 Instance-Level Dependence Analysis and Code Generation

We propose a fine-grained dependence analysis and code generation approach to address the limitations of the prior treatments to both explicit and implicit synchronizations in GPU-to-CPU translations. The approach is based on thread-level dependence graphs (TLDGs), a kind of representation of the dynamic instances of the instructions in a GPU kernel with intra- and inter-thread dependences captured.

In this section, we first introduce the concept of TLDG, and then describe its usage for enabling efficient and correct code generation in GPU-to-CPU translations.

Without loss of generality, we assume that the target code region for our following analysis meets the following two conditions: (1) It contains no loops; (2) the execution patterns of all thread blocks on that region are identical or the region is executed by only one thread block. These assumptions help focus our discussions on the handling of the synchronization issues; the elided complexities can be handled by existing compiler techniques without major adjustments to our analysis framework.

[1] We use source code rather than assembly instructions for illustration purpose.

4.1 TLDG

A TLDG is a directed graph constructed based on the data and control dependences in the GPU code, with awareness of the semantics of the warp/block logical hierarchy and synchronizations. It captures the important dependences that will appear in the execution of a GPU thread block. Depending on the number of threads a TLDG models, a TLDG can be for a thread block or a thread warp. The former is useful for dealing with explicit synchronizations, while the latter is for implicit synchronizations.

The generation of the node set of TLDG focuses on only those statements that access data (e.g. arrays) shared by different threads in the warp or block. The first step is to break the statements into load/store references, as illustrated in Figure 3 (b). This step yields a set of data reference units (DRU), each containing exactly one reference to shared data. DRUs are the basic scheduling units in the follow-up optimizations.

The second step is to build up a set of static nodes, as illustrated by the nodes in Figure 3 (c). Each DRU maps to one static node. Each node is marked by the array reference in its corresponding DRU. An instruction that accesses no shared data are attached to a node whose corresponding DRU follows that instruction.

The third step creates the nodes in the TLDG by duplicating the entire set of static nodes N times (N is the number of threads to model). Each node corresponds to the dynamic instance of a static node executed by a GPU thread.

The final step connects the nodes via directed edges, as shown in Figure 3 (d). Each edge $n_1 \rightarrow n_2$ represents one of the following two possibilities:

– There is a control dependence from n_1 to n_2, when both nodes come from the same thread, or
– There is a data dependence from n_1 to n_2 coming from either the same or different threads, where the type of data dependence could be either true, anti- or output dependence.

Dependence analysis is done between every pair of DRU. Due to the regular data-level parallelism of many GPU kernels, compiler-based static analysis is often sufficient.

Note that nodes constructed from the same DRU are executed simultaneously on GPU. And because there are no loops in the code, dependence edges can not point from a later DRU to an earlier DRU. Thus if we organize the nodes in a matrix, with the thread id increasing along the horizontal direction and the time stamp of each DRU execution increasing along the vertical direction, then no edges can point upwards, and no cycles exist in a TLDG.

The edges in an TLDG essentially compose the set of order constraints for the execution of the tasks in the GPU kernel. As long as a GPU-to-CPU translation of the kernel observes these constraints, the produced CPU code meets the order requirement. Next, we describe how we select a good instruction order among many legal schedules.

```
if (blockSize >=  8)  sdata[tid] += sdata[tid +  4];
if (blockSize >=  4)  sdata[tid] += sdata[tid +  2];
if (blockSize >=  2)  sdata[tid] += sdata[tid +  1];
```

(a)

```
//node 0
if (blockSize >=  8)  {tempBuf[tid] =  sdata[tid+4];}
//node 1
if (blockSize >=  8)  {sdata[tid] =+  tempBuf[tid];}
//node 2
if (blockSize >=  4)  {tempBuf[tid] =  sdata[tid+2];}
//node 3
if (blockSize >=  4)  {sdata[tid] +=  tempBuf[tid];}
//node 4
if (blockSize >=  2)  {tempBuf[tid] =  sdata[tid+1];}
//node 5
if (blockSize >= 2)  {sdata[tid] +=  tempBuf[tid];}
```

(b)

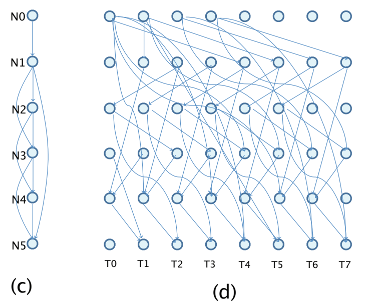


Fig. 3. (a). The original statements in CUDA SDK source code. (b). Statements broken into references, each forming a DRU. (c). The static nodes and dependences. (d) The intra-thread and inter-thread edges of the TLDG.

4.2 Code Generation

To generate a piece of CPU code from the TLDG, we need to introduce additional ordering between the DRUs without changing any source-sink relationships of the original GPU code. As the graph is acyclic, a simple breadth-first traversal of the graphs will yield a correct sequence. The rest part of the problem is comparing the quality of all the legal sequences and picking an optimal one. The produced code will be a sequence of all DRU instances, and no loops are needed to express the operations of a thread block.

Our code generation algorithm is a round-based scheduling algorithm as shown in Figure 4. The key idea is to partition the nodes into different groups and impose strict order among groups while maintaining full concurrency within each group. The DRUs in each group forms one round. In each round, the algorithm puts all nodes in the current TLDG that have no incoming edges into the round group (roundQueue in Figure 4), and then removes them along with their outgoing edges from the TLDG before proceeding to the next round. The process continues until the graph is empty.

A simple output of the instructions contained in each round in the round order will produce a correct execution sequence of DRUs. It is easy to see that the produced code maintains the source and sink relationships of all the dependences in the original TLDG. The successful detection and preservation of instance-level dependences effectively eliminate the need for a whole block syn-

```
while G not empty:
      for each node N :
            if N.inDeg == 0
               roundQueue.push(N);
               for each edge E outgoing from N:
                   delete E from G
               delete N from G
      roundQueue.sort(N)
      outputCode.append(roundQueue.codeGeneration())
```

Fig. 4. Pseudo code for round-based code generation

chronization. Such relaxation introduces additional freedom in the optimization space for GPU-to-CPU code generation. Figure 5 (a) shows the content of a piece of generated code.

This round-based code scheduling shares some commonality with traditional list-based instruction scheduling [7]. A key difference is that the round-based algorithm works on dynamic instances of instructions (TLDG) rather than static instructions (as in traditional instruction dependence graphs).

A simple optimization is to compute the values of thread ID-specific conditional statements during the code generation process to save runtime execution of those branch statements.

Another by-product of the above code generation process is the change of memory-access pattern in the original GPU program. Since memory coalescing and layout transformation are often explicitly maintained by GPU programmers, we would normally expect the memory referencing code of the GPU program to produce relatively regular memory accesses. Therefore the unrolling of the original loop into CPU code might impair the sequentiality and locality of memory accesses. To alleviate this problem, we simply add a sorting process within each round to heuristically reduce the spatial distance between two adjacent references in the generated CPU code. A more detailed study is part of our future studies.

4.3 Instance-Level Redundancy Removal

Due to the massive parallelism of GPU and the sensitivity of its efficiency to conditional branches, it is common for a GPU application to contain some useless calculations. Such redundancy differs from the redundancy in traditional CPU code in that the corresponding statements are not completely redundant. It is common that the executions of them by some but not all threads are useful. An example is the bottom line in Figure 2 (b). Even though only the calculation by the first thread is useful, all threads in the thread block runs that statement. The execution results of these threads are ignored automatically, causing no harm to the GPU computing efficiency.

But they may impair the CPU efficiency substantially. Take parallel reduction as an example. As a fundamental parallel algorithm that produces relatively small amount of data from large number of input entries, parallel reductions are often implemented under the rationale of reducing the length of the critical path as much as possible rather than the utilization of the processor. A typical parallel reduction code taken from CUDA SDK shows that no iterations after the first one actually utilizes more than half of all threads, but the redundant threads perform calculations just like the small portion of useful threads, creating large waste of processor time that can not be effectively hidden on CPU.

With instance-level computations fully exposed, the TLDG described earlier offers the basis for finding such redundant calculations. As shown in figure 5, the TLDG-based redundancy removal starts from a list of “useful” nodes and traces upwards to the top of the graph, marking all the nodes encountered in the process as useful. The initial set of useful nodes are those containing returning values of the kernel.

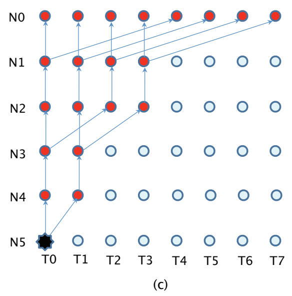
After redundancy removal, the number of lines of code generated for the reduction kernel (when block size is 256) is reduced from more than 3000 to around 500, significantly cutting the number of instructions CPU needs to execute.

```
   1 {
   2 T*data,
   3 hash<int, T> tempBuf)
   4 if(tid[128]<256) tempBuf.insert(<0>, data[128]);
   5 if(tid[129]<256) tempBuf.insert(<1>, data[129]);
......
3458 if(tid[239]<32) data[239] += tempBuf.pop(239);
3459 if(tid[255]<32) data[255] += tempBuf.pop(255);
3460}
```

(a)

```
   1 {
   2 T*data,
   3 hash<int, T> tempBuf)
   4 tempBuf.insert(<0>, data[128]);
   5 tempBuf.insert(<1>, data[129]);
......
511  data[1]+=tempBuf.pop(1);
512  tempBuf.insert(<0>, data[1]);
513  data[0]+=tempBuf.pop(0);
514 }
```

(b)



(c)

Fig. 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.

Similar to the optimization in code generation, redundancy removal can also be integrated into the code generation framework. With the large proportion of unnecessary memory references and conditional checks removed, the pruned code may outperform the original code substantially.

Our discussions so far concentrate on block-level TLDG for handling explicit synchronizations and block-level redundant calculations. It is easy to see that the same approach applies to warp-level implicit synchronizations and redundancies. The only change needed is to replace the block-level TLDG with a warp-level TLDG.

5 Evaluation

In this section, we present experiment result using the TLDG framework on 3 benchmarks: `reduction` and `sortingNetwork` from the CUDA SDK examples, and the CUDA version of the `NPB CG` benchmark, a conjugate gradient application [1]. All three benchmarks demonstrate both explicit and implicit synchronizations.

To test the performance of our framework, our experiment was carried out on a quad-core Intel Xeon E5460 machine, with Linux 2.6.33 and GCC 4.1.2 installed. The compilations always use the highest level of optimization.

5.1 Versions

For each benchmark, we create five versions for comparison.

- *Baseline:* This version is generated by MCUDA, a typical existing GPU-to-CPU translator available to the public. Note that due to the negliction of implicit synchronizations, the translation results from this version may not be correct.
- *Merged Version:* This version is a simple extension of MCUDA. It addresses the implicit synchronization issue by treating them as explicit synchronizations—that is, the statements between every pair of implicit synchronizations becomes a separate loop. We then employ the loop fusion in existing C compilers to reduce the incurred loop overhead.
- *Split Version:* This version is produced by a statement-level dependence analysis proposed in a recent study [12]. It applies a dependence theory to identify critical implicit synchronization points in a kernel and conducts loop splitting at those points to handle implicit synchronizations. This approach may avoid the drawbacks of the merged version in creating too many small loops.
- *TLDG-basic Version:* This version is from our basic approach. It is based on instance-level dependence analysis as described in earlier sections. Besides handling implicit synchronizations correctly, it relaxes the order constraints imposed by both implicit and explicit synchronizations through the round-based scheduling.
- *TLDG-opt Version:* This version is the same as the TLDG-basic, but with the TLDG-based redundancy removal applied.

5.2 Experiment Results

Table 1. Relative Speedup over the (incorrect) baseline version

Versions	Reduction	Sorting	CG
Baseline	1	1	1
Merged	0.38	0.72	0.93
Split	0.63	1.01	0.98
TLDG-basic	1.34	1.58	3.16
TLDG-opt	1.61	1.58	12.47

In our experiment, the timing results correspond to the entire-kernel execution for `reduction` and `sortingNetworks`, while for `CG-CUDA`, the it corresponds to the time spent in the two reduction bodies on the *common* array.

In Table 1, we compare the performance of the five versions. The merged version lags behind all other versions with considerable slowdown. It is due to the high loop overhead introduced by the transformation and the limited capability of compilers in loop fusion. The split version always demonstrates similar performance as the baseline, proving the effectiveness of statement-level dependence analysis and the moderate overhead of the synchronizations inserted upon the analysis.

The TLDG-based version outperforms both merged and split versions significantly in two of the three benchmarks, even without the redundancy removal. The main reason for such an advantage is its fully unrolled instruction sequence, which has almost no loop overhead, and meanwhile, provides a chunk of linear code for compiler to optimize. The locality produced by the intra-round sorting may also contribute to the speedup to a certain degree.

One exception is `reduction`, where TLDG-based version shows the worst performance among all versions. The reason lies in the implementation details. Since the original loop structure is broken and then fused into a bigger function body, adjacent DRUs from the same GPU thread might be separated by a large number of instructions from other threads. To avoid introducing unnecessary variables renaming, we used a temporary buffer in the generated code to store the intermediate results of each DRU, as well as its own thread id to cope with the frequent condition calculations in the `reduction` kernel. As shown in figure 5 (a) and (b), this buffer is implemented as a hash table to enable rapid loop-up for the latest stored value of a particular GPU thread. Such a design introduces some additional memory accesses. With only two explicit synchronizations per kernel invocation in the `reduction` benchmark, the time saving from enlarged basic block in the CPU code is not sufficient to outweigh this overhead. In CG, the synchronizations are repeated in a loop, while in `sortingNetworks`, there are a large number of memory loads and stores from the swapping process. Both provide sufficient opportunities for the compiler to take advantage of.

The TLDG-opt version gives the best performance of all. The speedup comes from the downsized CPU code with all useless operations and condition calculations removed. On the CG benchmark, thanks to is large kernel size and block size, the speedup is the most prominent. `Reduction` also shows a significant speedup. The code size of these two benchmarks are reduced by a factor of 8 and 6.8 respectively, compared with their TLDG-basic versions. The program, `sorting`, shows no extra speedup as it contains no redundancies.

6 Related Work

A number of previous works have aimed at automatic compilation of GPU program onto CPU. MCUDA [9, 10] and Ocelot [8]) both use an iterative execution framework based on the original GPU code structure to take advantage of its data and logical regularities. However, neither of them addresses the implicit synchronization pitfall, nor relaxes the constraints imposed by GPU synchronizations.

A recent study [12] analyzes the correctness issue caused by the negliction of implicit synchronizations. It proposes a state-level dependence theory to identify critical implicit synchronizations and generate correct CPU code. This current study concentrates on instance-level analysis and transformations.

NVIDIA provides a native emulation tool for running CUDA programs on CPU focuses on easing the debugging on GPU rather than improving performance [2]. Under the emulation mode, the programmer needs to manually insert macros to adapt to the current device at runtime. Although CUDA emulator provides the capability to run GPU program on CPU, it is not suitable for GPU-to-CPU automatic compilation. A similar case lies in OpenCL. While it allows the use of implicit synchronizations, it does not specify how they should be treated differently on different platforms, and the programmer again has to manually ensure the correctness of the cross-platform compilation [3]. Neither of them has attempted to relax order constraints or remove redundancies.

There have been many studies trying to ease GPU programming. A common approach is pragma-guided translation (e.g., from OpenMP to CUDA) [4, 14]. Others have proposed extensions to CUDA or OpenCL (e.g. [16]). Dynamical optimization of GPU executions through either software (e.g. [5, 6, 13, 17, 19–21]) or hardware (e.g. [11, 15, 18]) techniques have shown large benefits recently. This current study is unique in focusing on the translation of synchronizations across devices.

7 Conclusion

In this paper, we examine the impact of explicit and implicit synchronizations on the compilation of GPU code to CPU. We propose an instance-level dependence analysis to help produce correct CPU code with efficiency optimized. The new approach employs TLDG to capture the dependences among dynamic instances of instructions of all threads in a thread block or warp. Assisted with round-based code generation and redundancy removal, it not only addresses the correctness issue in the treatment to implicit synchronizations by existing techniques, but also leads to significant speedups on three benchmarks.

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 under Grant No. 0811791 and 0954015. 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.

References

1. Hpcgpu project. http://hpcgpu.codeplex.com/.
2. NVIDIA CUDA Programming Guide. http://developer.download.nvidia.com.
3. OpenCL. http://www.khronos.org/opencl/.
4. E. Ayguade, R. M. Badia, D. Cabrera, A. Duran, M. Gonzalez, F. Igual, D. Jimenez, J. Labarta, 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, Berlin, Heidelberg, 2009. Springer-Verlag.
5. 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.
6. S. Carrillo, J. Siegel, and X. Li. A control-structure splitting optimization for gpgpu. In *Proceedings of ACM Computing Frontiers*, 2009.
7. K. Cooper and L. Torczon. *Engineering a Compiler.* Morgan Kaufmann, 2003.
8. G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems. In *Proceedings of The Nineteenth International Conference on Parallel Architectures and Compilation Techniques*. ACM, 2010.
9. J. A. S. et. al. Mcuda: An ecient implementation of cuda kernels for multi-core cpus. In *LCPC'08*, 2008.
10. J. A. S. et. al. Efficient compilation of fine-grained spmd-threadedprograms for multicore cpus. In *CGO'10*, 2010.
11. 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.
12. 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.
13. A. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: Portable stream programming on graphics engines. In *ASPLOS'11*, 2011.

14. S. Lee, S.-J. Min, and R. Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. In *PPOPP'09*, pages 101–110, 2009.
15. J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *ISCA'10*, 2010.
16. S. Michel, K. Philipp, and G. Sergei. Skelcl - a portable skeleton library for high-level gpu programming. In *IPDPS'11*, 2011.
17. 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.
18. D. Tarjan, J. Meng, and K. Skadron. Increasing memory latency tolerance for simd cores. In *SC'09*, 2009.
19. Y. Yang, P. Xiang, J. Kong, and H. Zhou. A gpgpu compiler for memory optimization and parallelism management. In *PLDI*, 2010.
20. 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.
21. E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *ASPLOS'11*, 2011.