



Dissertations, Theses, and Masters Projects

Theses, Dissertations, & Master Projects

2011

Treatment of Synchronizations in Compiling Fine-Grained SPMD-Threaded Programs for CPU

Ziyu Guo

College of William & Mary - Arts & Sciences

Follow this and additional works at: <https://scholarworks.wm.edu/etd>

 Part of the [Computer Sciences Commons](#)

Recommended Citation

Guo, Ziyu, "Treatment of Synchronizations in Compiling Fine-Grained SPMD-Threaded Programs for CPU" (2011). *Dissertations, Theses, and Masters Projects*. Paper 1539626921.

<https://dx.doi.org/doi:10.21220/s2-90a6-pm70>

This Thesis is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

Treatment of Synchronizations in Compiling Fine-Grained SPMD-Threaded
Programs for CPU

Ziyu Guo

Wuhan, Hubei, China

Bachelor of Engineering, Harbin Institute of Technology, 2006

Master of Engineering, Chinese Academy of Sciences, 2009

A Thesis presented to the Graduate Faculty
of the College of William and Mary in Candidacy for the Degree of
Master of Science

Department of Computer Science

The College of William and Mary
August 2011

APPROVAL PAGE

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

Master of Science

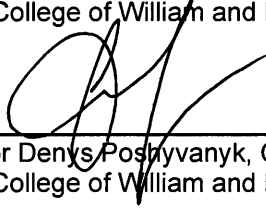


Ziyu Guo

Approved by the Committee, July, 2011



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



Assistant Professor Denys Poshyvanyk, Computer Science
The College of William and Mary



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

ABSTRACT PAGE

With the industry rapidly transiting into multicore/manycore era, heterogeneous systems will be the mainstream in the foreseeable future, and thus requires a highly versatile compilation framework that is able to generate efficient code for different architectures in the system from a single version of source code. However, the device-specific programming models on these devices make such translation difficult. A prominent exhibition of the difficulty exists in the compilation of fine-grained SPMD-threaded code (e.g., GPU CUDA code) for multicore CPUs.

In this thesis we propose a reference level dependence analysis algorithm to reveal the relationships between the correctness and performance of the translated program and the dependencies introduced by implicit synchronizations. Based on the analysis result we present several low-overhead extensions to previous GPU-CPU compilation schemes with guaranteed correctness and improved performance. To utilize the instance-level dependence information, we propose thread-level dependence graph (TLDG), which leads to a method that enables fine-grained treatment to both implicit and explicit synchronizations, and reveals redundant computation at the instruction-instance level. We then present an automatic framework that performs such treatment on GPU code.

Together, the dependence analysis and code generation schemes form a complete solution to the problem of GPU-to-CPU translation of synchronizations for the first time. The methods presented in this thesis can act as basis for treating other device-specific intrinsics, and is critical for the whole-system synergy in heterogeneous systems.

Table of Contents

Dedication	iii
Acknowledgements	iv
List of Tables	v
List of Figures	vi
1 Introduction	2
2 Problem Analysis	6
2.1 Background on CUDA and SPMD-Translation	6
2.2 A Correctness Pitfall	9
2.3 Error from Insufficient Preservation of GPU Threads' Mask	11
2.4 Efficiency Issue	11
3 Dependence Analysis	15
3.1 Coarse Grained Analysis	15
3.2 Fine Grained Analysis	21
3.2.1 Construction of TLDG	22
4 Solutions	25
4.1 Coarse-Grained Solution	25
4.1.1 Solution 1: A Dependence-Based Splitting-Oriented Approach	25

4.1.2	Solution 2: A Merging-Oriented Approach	30
4.2	Fine-Grained Solution	30
4.2.1	Instance-level Redundancy Removal	33
4.2.2	Discussion	34
5	Evaluation	36
5.1	Coarse-Grained Solution	36
5.1.1	Methodology	36
5.1.2	Experimental Results	37
5.2	TLDG-Based Solution	40
5.2.1	Versions	41
5.2.2	Experiment Results	42
6	Related Work and Conclusion	45
6.1	Related Work	45
6.2	Conclusion	46
	Bibliography	48
	Vita	51

✓ Dedicated to my parents.

ACKNOWLEDGMENTS

This thesis would not have been possible without the support of many people. I would first attribute this thesis to my adviser Dr. Xipeng Shen. I sincerely thank him for the invaluable guidance, advises and remarks he has given me during the past 2 years. I have always admired his level of concentration and dedication, have been encouraged by his passion and inspired by his keen wit.

I also want to mention my colleagues and friends at William and Mary for their support for me, both academic and personal, during the course. Eddy Zheng Zhang, Yunlian Jiang and Kai Tian have been immensely helpful in my research.

I am grateful to the faculty and staff in the Computer Science department for their assistance over the years.

Most importantly, I would like to express my deepest gratitude to my family for their support over all these years, which I can never repay in equal. Your love is what drives me ahead.

List of Tables

5.1	Benchmarks	37
-----	----------------------	----

List of Figures

2.1	Illustration of MCUDA compilation.	8
2.2	Parallel reduction with implicit synchronizations used. (Assuming warp size=32, block size>= 64.)	10
2.3	Original GPU thread mask modified during cpu iterations.	12
3.1	Examples for the reverse postorder (rpn) of basic blocks and the sequence numbers (enclosed by “[]”) of instructions.	16
3.2	Examples for demonstrating the SPMD-Translation Dependence Theorem. The code segments (a) to (i) are examples of GPU kernel code. The captions show the dependence sign vectors of their <u>corresponding CPU code</u> produced by the basic SPMD-translation, as illustrated by graph (j). Only the dependences in graphs (g,h,i) are critical for SPMD-translation. (Loops are assumed to have been normalized with indices increasing by 1 per iteration; elided code has no effects on dependences.)	20
3.3	(a). The original statements in CUDA SDK source code. (b). Statements broken into references, each forming a DRU. (c). The intra-thread and inter-thread edges of the TLDG constructed from (b).	23
4.1	Algorithm for step 5 in Solution 1.	27
4.2	An example for Solution 1.	28
4.3	Illustration of translating a GPU loop with thread-dependent critical implicit synchronizations into CPU code.	29

4.4	Pseudo code for round-based code generation	30
4.5	(a). The original hardcode without redundancy removal. (b). Pruned hardcode where all useless computations are removed. (c). The bottom-up redundancy removal process, start from the compiler identified useful final results. (marked black)	34
4.6	The original TLDG (upper left) broken down into 6 basic patterns, each of which retains its shape and orientation in the whole graph, only repeated on the horizontal direction.	35
5.1	Running times on the Intel machine, normalized to the execution times of the (erroneous) basic SPMD-translation results. (“sortNet” is SortingNetworks in short)	39
5.2	Running times on the AMD machine, normalized to the execution times of the (erroneous) basic SPMD-translation results. (“sortNet” is SortingNetworks in short)	40
5.3	Relative performance compared to (incorrect) baseline version.	42

Treatment of Synchronizations in Compiling Fine-Grained
SPMD-Threaded Programs for CPU

Chapter 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, making heterogeneous systems increasingly popular. With the spectrum of applications being ported onto accelerators becomes broader, more efforts have also been invested into developing specialized code for the explicitly parallel, fine-grained SPMD-threaded execution model on GPU.

However, the rapid adoption of GPU-specific programming models, such as NVIDIA CUDA, brings the challenge of programming such systems. Purely relying on these device-specific models would require the development of separate code versions for different devices. It not only hurts the programmers productivity, but also creates obstacles for code portability, and adds restrictions for using cross-device task migration or partition to promote whole-system synergy. Therefore recent years have seen a number of efforts trying to develop a single programming model that applies to various devices. These efforts include development of new programming languages (e.g., Lime [6]), libraries (e.g., OpenCL [4]), and cross-device compilers (e.g., CUDA Fortran compiler [25], *O₂G* [17], MCUDA [13, 12], Ocelot [11]).

GPU-to-CPU translation aims at compiling code written in these programming models to CPU code. 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. Even though the GPU-to-

CPU translation may not be the ideal route to reaching the goal of “one-code-fits-all”, it is important, given that the number of applications written in GPU-specific programming models increases continuously. Second, the translation enables smooth collaboration between CPU and GPU processors. 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 the whole system synergy for the execution of a GPU application.

In a fine-grained SPMD-threaded program, a large number of threads execute the same kernel function on different data sets; the task of a thread is in a small granularity, hence parallelism among tasks are 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. A major problem with all previous translations schemes is that none of them has systematically explored the different implications of device specific intrinsics on GPU. These intrinsics often help utilize the unique hardware function units on the accelerator, or eases the implementation of various parallel operations, and therefore are used prevalently. In CUDA [2], block level synchronization (`__syncthreads()`) is one of the most widely used device intrinsics. Its acts as a block level barrier, stalling each thread in the block until all have reached synchronization point. The very low overhead [22] of this intrinsics makes it favorable to programmers, and often used as an easy and conservative implementation whenever there exists dependences between statements. By doing this the programmer actually enforces unnecessarily strong constraints in the GPU program. It causes almost no issue on GPU 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; that practice causes even more serious issues: The produced CPU code may be semantically erroneous because of the violation of some data dependences maintained by the implicit synchronizations in the original GPU code (an example is shown in Section 2.2).

In this thesis, we conduct a systematic study on the issue, particularly in the context of compiling fine-grained SPMD-threaded programs (called *SPMD-translation* in short) for multicore CPU. We discuss the origin, forms, and performance implications of GPU synchronization intrinsics, both the implicit (Section 2.2) and explicit (Section 2.4). We point out a correctness pitfall current SPMD-translations are subject to. By analyzing the impact of inter-thread data dependences and intrinsics' semantics in GPU-CPU translation, we present a comprehensive framework to provide correct and efficient translation,

Our study uses CUDA as the fine-grained SPMD-threaded programming model for its broad adoption. We show that the treatments in current SPMD-translation to implicit synchronizations are insufficient to guarantee the correctness of the produced programs (Section 2.2). Through dependence analysis, we reveal the cause of the compromise to correctness and efficiency and the relations with various types of dependences in a program (Section 3.1).

Based on the findings, we then develop three solutions (Section 3.1, Section 3.2). The *first* is a splitting-oriented approach, which starts with the (possibly erroneous) compilation result of traditional SPMD-translation, and tries to fix the translation errors by detecting critical implicit synchronization points, and splitting the code accordingly. The *second* solution is based on simple extensions to prior SPMD-translations. It is merging-oriented. It treats implicit synchronizations as explicit ones, uses the prior SPMD-translations to

produce many loops containing one instruction each, and then relies on standard compilers to reduce loop overhead through loop fusion. We add some remedies to make it handle thread-dependent synchronizations. The third is based on thread-level dependence graphs (TLDG) (Section 3.2) and extracts dynamic fine-grained data and control dependences. It relaxes unnecessary synchronization constraints of both kinds, and prunes instruction-instance-level redundant computations to improve the efficiency of the generated CPU code. This solution is an universal treatment for translating synchronizations on GPU.

We evaluate the techniques on a set of programs that contain non-trivial implicit or explicit synchronizations (Section 5.1). The results show that the proposed dependence analysis and solutions resolve the correctness issue in existing SPMD-translations effectively, with correct and efficient code produced for all tested benchmarks.

Overall, this thesis makes the following main contributions:

- This thesis reveals, for the first time, the impact of implicit and explicit synchronization on correctness and performance during the compilation of fine-grained SPMD-threaded programs onto CPU, and discusses the limitations of previous GPU-CPU compilation methods resulting from over-simplified treatment of implicit synchronizations and excessively strong constraints on explicit synchronizations.
- Based on this observation, this thesis proposed 2 levels of solutions: a set of coarse grained dependence analysis based extensions that resolves the correctness issue, and a unified TLDG-based fine-grained dependences analysis and translation framework that both guarantees correctness and brings extra performance gains as well as optimization opportunities.

Chapter 2

Problem Analysis

2.1 Background on CUDA and SPMD-Translation

This section provides some CUDA and SPMD-translation background that is closely relevant to the correctness and efficiency issue uncovered in the following sections of this chapter.

Overview of CUDA CUDA is a representative of fine-grained SPMD-threaded programming models. It was designed for programming on GPU, a type of massively parallel device containing hundreds of cores. CUDA is mainly based on the C/C++ language, with several minor 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.

Explicit and Implicit Synchronizations on GPU On GPU, there are mainly two types of synchronizations. Explanations of them relate with GPU thread organization.

GPU threads are organized in a hierarchy. 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*. Execution and synchronization models differ at different levels of the hierarchy. 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 has finished the current instruction. Such a kind of synchronizations are called **implicit synchronizations**, as no statements are needed to trigger them; they are enabled by hardware automatically. There is another type of synchronization. By default, different warps run independently. CUDA provides a function “`_syncthreads()`” for cross-warp synchronizations. The 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 the barrier. Such synchronizations are called **explicit synchronizations**. In CUDA, there is no scheme (except the termination of a kernel) for enabling synchronizations across thread blocks.

It is worth noting that in CUDA, control flows affecting an explicit synchronization point must be thread-independent—that is, if the execution of a synchronization point is control-dependent on a condition, that condition must be thread-invariant. In another word, “`_syncthreads()`” cannot appear in a conditional branch if only part of a thread block follows that branch. This constraint, however, does not apply to implicit synchronizations: They exist between every two adjacent instructions; there is no exception. This difference causes some complexities for treating implicit synchronizations by simply extending current solutions to explicit synchronizations, as we will show in Section 3.1.

SPMD-Translation The goal of SPMD-translation is to compile fine-grained SPMD-threaded programs to code acceptable by other types of devices. MCUDA [13, 12] is a recently developed compiler for SPMD-translation. For its representativeness, we will use it as the example for our discussion.

MCUDA is a source-to-source compiler, translating CUDA code to C code that run

on multicore CPU. Its basic translation scheme is simple. For a given GPU kernel to be executed by N_B thread blocks, MCUDA creates N_B parallel tasks, with each corresponding to the task executed by a thread block in the GPU execution of the program. A generated parallel task is defined by a C function (called a CPU task function), derived from the GPU kernel function: Each code segment between two adjacent explicit synchronization points (including the beginning and ending of a kernel) in the GPU kernel function becomes a serial loop in the CPU task function. Each of such loops has B iterations (B is the number of threads per GPU thread block), corresponding to the GPU tasks of a thread block. Figure 2.1 shows an example (with some simplifications for illustration purpose).

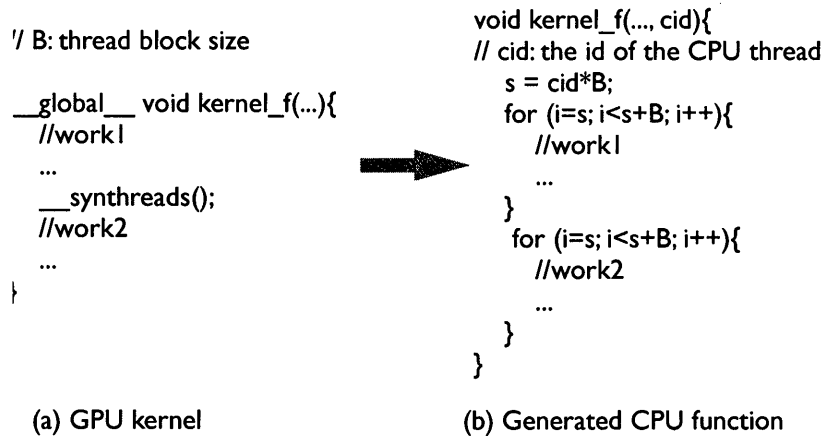


Figure 2.1: Illustration of MCUDA compilation.

It is easy to see that the translation keeps the semantics of explicit synchronizations: No instruction after a synchronization point (e.g., the second loop in Figure 2.1) can run until all instructions before the synchronization point (e.g., the first loop in Figure 2.1) have finished. MCUDA gives appropriate treatment to local and shared variables, branches (e.g., *break*, *continue*, etc.), loops, and some other complexities in a kernel. In a parallel execution on CPU, the N_B parallel tasks will be assigned to CPU threads appropriately to achieve high performance.

From now on, we call the SPMD-translation represented by MCUDA as the *basic SPMD-translation*. As seen, MCUDA ensures correct treatment to explicit synchronizations in

a kernel through loop fission. However, as all existing SPMD-translation tools, MCUDA ignores implicit synchronizations in a kernel, which may cause erroneous translation results, as discussed next.

2.2 A Correctness Pitfall

We first use a simple, contrived example to explain the correctness issue that current SPMD-translations are subject to because of implicit synchronizations.

Suppose that the "work1" in Figure 2.1 contains the following statement

S1: `if (tid < warpSize) {A[tid] += A[tid+1]; B[tid+1] = A[tid+1];}`,

where, *tid* is the ID number of the current GPU thread. In the default MCUDA compilation, this statement will remain unchanged in the generated code (Figure 2.1 (b)) except that the *tid* will be replaced with the thread loop index variable *i*.

Recall that threads in a warp proceed in an SIMD manner. So for statement S1 in a GPU execution, no instance of "B[tid] = A[tid+1]" will be executed until all instances of "A[tid] += A[tid+1]" finish. The implicit synchronization between the two statements hence ensures that the updates to the elements in *B* (except $B[\text{warpSize} + 1]$) come from the new values of *A*. However, because MCUDA neglects the implicit synchronization, the generated CPU code fails to maintain the semantics: Each iteration of the first loop would copy the *old* value of an element of *A* to *B*.

Such a reliance on implicit synchronizations appears in some commonly used GPU applications. An example is the parallel reduction program in the CUDA SDK [3]. It computes the sum of an input array. The execution of a thread block computes the sum of a chunk in the input array. The algorithm is the classic tree-shaped parallel reduction algorithm, as shown in Figure 2.2 (a). Each middle level of the tree corresponds to one step in the reduction and computes the partial of the sum.

Figure 2.2 (b) shows a piece of code from the GPU kernel of the reduction program in CUDA SDK. Each iteration of the "for" loop corresponds to the reduction at one level of

the tree. Because of the dependences between levels, an explicit synchronization appears at the bottom of the loop body.

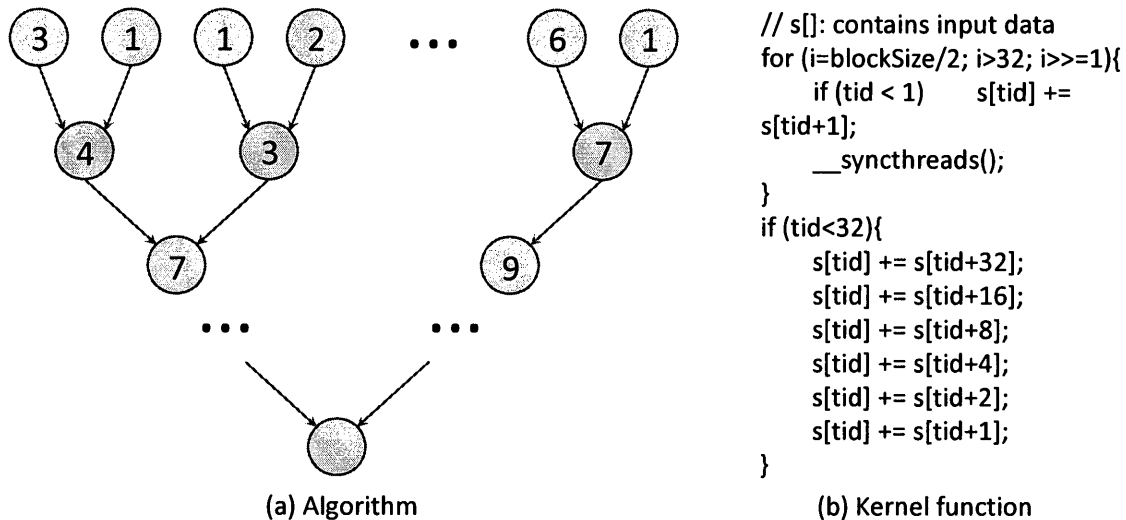


Figure 2.2: Parallel reduction with implicit synchronizations used. (Assuming warp size=32, block size \geq 64.)

The six lines of code below the "for" loop in Figure 2.2 (b) are for the bottom six levels of reduction. Even though dependences exist among these levels, there are no synchronization function calls among the six lines. This is not an issue because only the execution of the first warp matters to the final result and there are implicit intra-warp synchronizations already.

The motivation for GPU programming to leverage implicit synchronizations is computing efficiency. For instance, the way in which the final six levels of the reduction tree are implemented comes from optimizations. In an earlier version of the reduction in CUDA SDK, they are actually the final six iterations of the "for" loop (whose loop header is in a form "for (i=blockSize/2; i>0; i>>=1)"). The optimized form saves loop index computation, invocations to the explicit synchronization function, and unnecessary synchronizations across warps. These benefits yield 1.8X speedup as reported by NVIDIA [15].

Because of such large performance gains, similar exploitations of implicit synchronizations are common in some important, high-performance programs (e.g., sorting, reduction,

prefix-sum, etc.). Current SPMD-translations lack not only the capability to treat such synchronizations systematically, but also the functionality to detect such critical implicit synchronizations, hence jeopardizing their soundness and practical applicability.

2.3 Error from Insufficient Preservation of GPU Threads' Mask

In the scenario where different threads in a warp diverge, GPU threads execution are serialized and the sets of threads that follow each path have to be enumerated individually. Since SIMD model dictates that each thread executing exactly the same instructions, the threads not designated to enter a certain path will be masked off and therefore will not commit their execution result to the shared memory. On GPU, such mask is implemented in hardware and managed by CUDA runtime to ensure its correctness through out this process, and any instructions executed after the diverge point should not be able to modify the mask. However MCUDA incorrectly assumes that synchronization points are thread-independent, which holds for explicit synchronizations, but not for implicit synchronizations. Figure 2.3 exemplifies this problem. Consider that warp size is 2, and the initial values of A, B, P are $A = \{-1, 1\}$, $B = \{2, 2\}$, $P = \{1, 0\}$. In the original GPU execution, *only the second thread goes back to L and for only once*, the computing results are $A = \{1, 2\}$ and $B = \{1, 0\}$. But the execution of the CPU code will go back to L twice and produces results as $A = \{1, 3\}$ and $B = \{0, 0\}$.

2.4 Efficiency Issue

In MCUDA, the common approach to transforming GPU `__syncthreads()` into equivalent CPU code is to imitate the strict intra-block barrier via loop splitting. This approach relies on the 2-level nested loop structure created during the kernel transformation. While the outer grid level loop remains unaffected, the inner block level loop should be split exactly at the location of the `__syncthreads()` call. Hence each innermost loop contains only 1

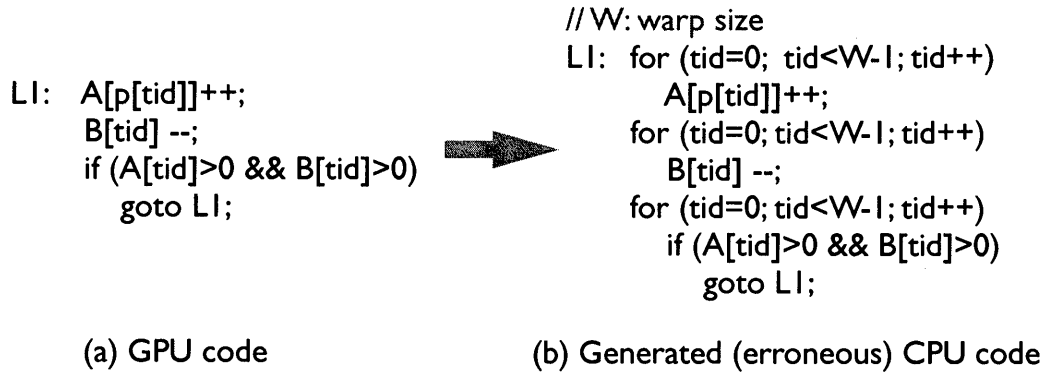


Figure 2.3: Original GPU thread mask modified during cpu iterations.

synchronization-free code block that iterates through all threads that need to execute it. Intuitively, this solution tries to maintain the execution order in the original GPU code on a coarse granularity, and its very slight code modification makes it both easy to implement and test. However, as discussed below and in following section, this loop splitting approach is neither efficient nor guaranteed correct.

Figure 2.1 illustrates the translation scheme implemented in MCUDA. The kernel bodies referred to as *work1* and *work2* in (a) are wrapped into CPU loops in (b), essentially replacing GPU thread scheduling with CPU loop iterations. Note the substitution of `__syncthreads()` with loop splitting.

The constraints followed by loop splitting approach is unnecessarily strict, and a closer investigation shows that there are much space for relaxation in the strictly imitated block level synchronizations. Some of the issues are:

- **Introduction of additional loop overhead.** The `__syncthreads()` takes advantage of the hardware barrier function units on GPUe. Loop splitting, on the other hand, creates a small loop that on only contains one synchronization-free code block for every block. When the grid size becomes large(and they often do), the linearly increasing loop overhead soon becomes significant.
- **Excessively strong synchronization.** The efficient synchronization and uniform SIMD execution model makes it attempting to skip fine-grained data dependence

analysis during the coding of GPU kernels, and insert `__syncthreads()` wherever it might be needed. This is a valid strategy in GPU programming simply because it's light-weight and usually there's no better ways to enforce sequentiality readily available. Under this rationale a large number of `__syncthreads()` calls are placed where there can be more flexible solutions if the same kernel is to be translated onto CPU. For example, in one of our benchmarks, `CG_CUDA`, 23 `__syncthreads()` invocations were used in a kernel with only 170 lines of code. Consider the loop structure that surrounds these synchronizations, the number of their dynamic instances at runtime is alarming.

- **Fixed loop iteration order.** While the CUDA runtime and hardware scheduler ensures the correctness of `__syncthreads()` stalled threads, the execution order of those threads are not defined, and is solely dependent upon runtime scheduling. Such dynamic design provides enough flexibility in the order of job instances execution, and is essential to better memory latency hiding and data locality. The loop splitting approach, however, dictates a static order job instances, and all spaces for further optimizations are lost.

The reason for all three issues result from the inappropriate treatment of synchronizations. For implicit synchronizations, the key is to understand the the difference in semantic implication against explicit synchronizations. Implicit synchronizations exist everywhere, hence the explosion of the number of created loops; implicit synchronizations can be thread-dependent, hence the second issue. For explicit synchronizations, the excessively strong constraints on GPU calls for a flexible and efficient translation that preserves only the exact necessary synchronization constraints intended by the GPU programmer

To accommodate these issues, it is important to have a scheme to identify the actual semantics of critical synchronizations and generate code maintaining the exact semantics without introducing too much overhead.

A systematic dependence analysis is important for meeting both conditions. Traditional

dependence analysis offers many insights, but are not directly applicable as they consider no relations between data dependence across SIMD thread groups and the semantics and properties of GPU synchronizations. We next present a systematic analysis of the relations, and then describe several derived solutions to both implicit and explicit synchronization problem.

Chapter 3

Dependence Analysis

In this chapter we employ reference level and instance-level dependence analysis techniques to identify implicit synchronizations that might pose a hazard in previous approaches, and discuss several solutions to resolve such hazards.

3.1 Coarse Grained Analysis

This section examines the relations between various dependencies and compilation correctness related to implicit synchronizations. The reveal of these relations lays the foundation for identifying and appropriately treating critical implicit synchronizations.

For simplicity of explanation, our discussion in this part concentrates on a segment of kernel code C that contains no explicit synchronizations. Explicit synchronizations are already handled by the basic SPMD-translation. Because implicit synchronizations only apply to threads within a warp, we will restrict our discussion to the execution of C by a warp.

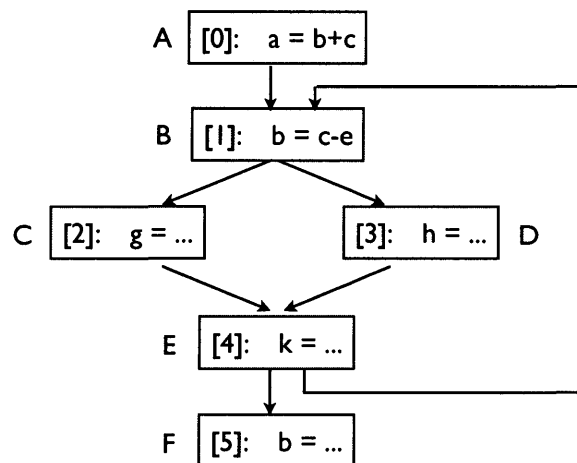
Our strategy for coarse-grained dependence analysis is to first use the default (problematic) SPMD-translation scheme, as described in Section 2.1, to derive a sequential loop L from C , and then conduct dependence analysis on L . This strategy circumvents the complexities in dealing with the multithreading behaviors in the original GPU code C .

From Section 2.1, we know that L essentially takes C as its loop body and adds a

surrounding loop for iterating through threads. (This loop is called a *thread loop*.) With only one warp considered, the loop index values span from 0 to $warpSize - 1$. All appearances of thread id in C are replaced with the loop index variable.

We say that L is correct if its executions on a CPU always produce the same results as the corresponding GPU executions of C do. Because L neglects all implicit synchronizations in C , instructions may be executed in an order different from the GPU execution of C , hence causing errors. Apparently, if there are no data dependences in L , there is no need to observe the implicit synchronizations: All execution orders produce the same results. Data dependences are the key factor for analysis.

Considering the properties of GPU executions, we introduce the following terms and notations (mostly derived from traditional terminology) to be used in our proposed dependence analysis.



$$\text{rpn}(A)=0; \text{rpn}(B)=1; \text{rpn}(C)=2; \text{rpn}(D)=3; \text{rpn}(E)=4; \text{rpn}(F)=5$$

Figure 3.1: Examples for the reverse postorder (rpn) of basic blocks and the sequence numbers (enclosed by “[]”) of instructions.

Terms and Notations

- *Reverse Postorder of Basic Blocks in L .* Following the traditional compiler terminology, we use *postorder* to refer to the order that basic blocks are last visited in

a depth-first search on the control flow graph of L . A *reverse postorder* is simply the reverse of a postorder. For SPMD-translation, however, we add the constraint that when the possible order of two blocks is not unique (e.g., sibling branches), the leftmost block has the precedence. (Without loss of generality, it is assumed that the CUDA compiler ensures that code block layout follows such a left-to-right order.) This constraint is useful for dependence analysis because the order in which a GPU thread warp traverses basic blocks, is consistent with this reverse postorder due to their SIMD execution mode. Roughly speaking, reverse postorder is a top-down order on a control flow graph but with branches and back-edges appropriately handled. We use $rpn(B)$ to represent the reverse postorder number of a basic block B . Figure 3.1 shows an example.

- *Sequence Number.* Each statement in L has a distinctive sequence number. Let S_1 and S_2 be two instructions in basic blocks B_1 and B_2 respectively, and n_1 and n_2 be the sequence numbers of the two statements. If $B_1 = B_2$, $n_1 < n_2$ if and only if S_1 precedes S_2 in the block. If $B_1 \neq B_2$, $n_1 < n_2$ if and only if $rpn(B_1) < rpn(B_2)$. An example is shown in Figure 3.1. We use $sn(S)$ for the sequence number of a statement S . The sequence numbers cover all instructions in L and gives them a single order that is consistent with the execution order of the instructions in GPU when back-edges are not considered (loops are treated through dependence vectors). Such an order offers conveniences for dependence analysis as shown later in this section.
- *Dependence Distance Vector.* This term is the same as in the traditional dependence theory [5]. Roughly speaking, it is the difference between the iteration vectors of two statements when they access the same memory location. Elements in an iteration vector (from left to right) corresponds to the loops enclosing the statement (from outermost to innermost). The value of an element is the index value of the corresponding loop. For example, the dependence distance vector from S_1 to S_2 in the right graph of Figure 3.2 (j) is $(1, -2, 1)$, where, the three elements correspond to the loops *tid*,

i , and j respectively. (It is important to note that the elements take the loop order rather than the array index order.) Only the loops enclosing both statements are considered in their dependence distance vectors.

- *Dependence Sign Vector.* It is just the results after a sign function is applied to the elements in a dependence vector. For instance, the dependence sign vector for the right graph of Figure 3.2 (j) is $(1, -1, 1)$. If there are multiple dependences between two statements and their dependence sign vectors differ, “*” can be used to represent the difference. For instance, two vectors $(-1, 0, 1)$ and $(1, 0, 1)$ can be represented with one $(*, 0, 1)$.¹
- *Preserved Dependence.* This term is identical to its traditional definition. A dependence between S_1 and S_2 is preserved after a transformation if the access order to common memory locations by the two operations remain the same as in the original program.
- *Critical Dependence.* A dependence is critical if it cannot be preserved after the basic SPMD-translation.
- $V(i : j)$. We use $V(i : j)$ to represent part of a vector (i.e., $V(i, i + 1, \dots, j)$), and use $V(i)$ for $V(i : i)$.

SPMD-Translation Dependence Theorem With the defined terms, we describe the following theorem, which offers the foundation for identifying critical dependences and implicit synchronization points for SPMD-translation. (Notations L and C have been defined at the beginning of this section.)

Theorem 3.1 SPMD-Translation Dependence Theorem: *Let S_1 and S_2 be two statements in L and $sn(S_1) < sn(S_2)$. Let d be a data dependence from S_1 to S_2 in C . Let v be the sign vector of the data dependence in L that corresponds to d . The dependence d is preserved in*

¹We use dependence sign vectors rather than traditional dependence direction vectors because the former is more intuitive and clear than the latter.

L if at least one of the following conditions holds:

- (1) $v(1) == 0$;
- (2) there are no non-zero element in $v(2 : |v|)$ & $v(1) < 0$;
- (3) $v(1)$ equals the first non-zero element in $v(2 : |v|)$ and that element is not “*”.

We now outline the proof of the theorem. We start with the first condition. The condition $v(1) == 0$ indicates that between S_1 and S_2 , there is no data dependence carried by the thread loop in L , which suggests that between S_1 and S_2 , there is no data dependence among threads in the execution of C . The neglect of the implicit synchronizations between the two statements in L hence affects no inter-thread data dependences. Graphs (c) and (d) in Figure 3.2 exemplify that the correctness holds regardless the remaining elements of v .

For the second condition, because there is no non-zero element in $v(2 : |v|)$, between S_1 and S_2 there must be no data dependences carried by any loop in C . Because of the SIMD execution mode and $sn(S_1) < sn(S_2)$, in one iteration of the common loops in C enclosing both S_1 and S_2 , executions of S_1 by all threads in a warp must finish before any execution of S_2 starts during the execution of C on GPU. Therefore, if there are data dependences, S_1 must be the source and S_2 must be the sink in the GPU execution of C . The condition $v(1) < 0$ ensures that the same dependence relation holds in the execution of L on CPU. Graphs (a) and (b) in Figure 3.2 illustrate such cases, while graph (g) shows a counter example.

To see the correctness of the third condition, we note that the appearance of non-zero elements in $v(2 : |v|)$ suggests that some loop(s) in C carries data dependences between S_1 and S_2 . The direction of the dependence during the execution of C on GPU is determined by the first non-zero element in $v(2 : |v|)$. While for L , it is the first non-zero element in v that determines the dependence direction between S_1 and S_2 in the execution of CPU. Therefore, the third condition ensures that the dependence direction remains the same between L and C . Graphs (e) and (f) in Figure 3.2 demonstrate that the correctness holds regardless the exact dependence directions between S_1 and S_2 , while Figure 3.2 (h) shows

a counter example.

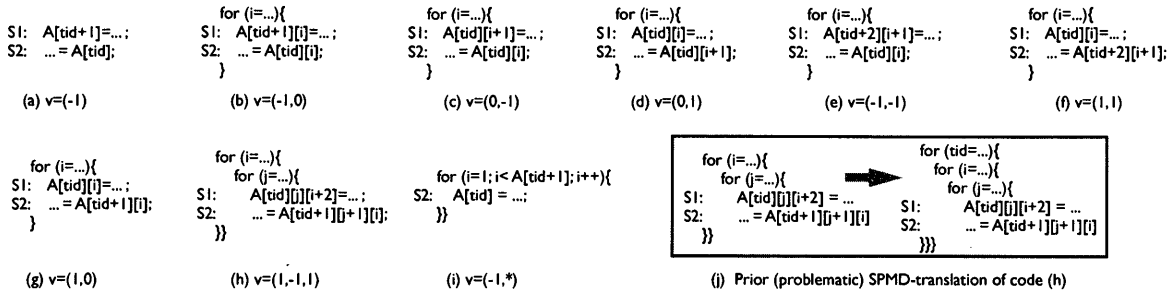


Figure 3.2: Examples for demonstrating the SPMD-Translation Dependence Theorem. The code segments (a) to (i) are examples of GPU kernel code. The captions show the dependence sign vectors of their corresponding CPU code produced by the basic SPMD-translation, as illustrated by graph (j). Only the dependences in graphs (g,h,i) are critical for SPMD-translation. (Loops are assumed to have been normalized with indices increasing by 1 per iteration; elided code has no effects on dependences.)

Two notes are worth mentioning. First, the theorem and proof do not distinguish locations where the dependence appears. So they hold regardless whether the dependence appears in a thread-dependent branch. For example, the statement S2 in Figure 3.2 (i) is in a thread-dependent branch—different threads in a warp may run the “for” loop for different numbers of iterations. The dependence sign vector is $(-1, *)$ from the loop conditional statement, “ $i < A[tid + 1]$ ”, to S2. It meets none of the three conditions in the theorem, indicating that such a dependence is critical and the basic SPMD-translation cannot preserve it.

Second, the SPMD-Translation Dependence Theorem mentions no dependence types. It is easy to see that the theorem holds no matter whether the data dependence is a true (read after write), anti- (write after read), or output (write after write) dependence.

Implications to SPMD-Translation The SPMD-Translation Dependence Theorem has three implications.

First, it facilitates the detection of SPMD-translation errors. Based on the theorem, a compiler will be able to examine a program generated by a basic SPMD-translation and tell whether it may contain data dependence violations.

Second, it lays the foundation for the detection of critical dependences and important implicit synchronization points (i.e., those affecting the correctness of the basic SPMD-translation), by revealing dependences meeting none of the three conditions. Section 3.1 will describe how this implication translates into a systematic detection scheme for critical implicit synchronizations.

Finally, the theorem provides the theoretical guidance for using loop transformations to fix certain errors in the basic SPMD-translations. For instance, as described earlier, the default SPMD-translation to the code in Figure 3.2 (g) yields a dependence vector $v = (1, 0)$, satisfying none of the three conditions, and hence indicating the error of the translation. However, it is easy to see that a simple reversal of the thread loop index in the CPU code turns the dependence vector into $v = (-1, 0)$, which meets the second condition of the theorem, and the dependence from S1 to S2 in the GPU code is preserved. Following chapter will show how this implication can be systematically exploited during code generation in SPMD-translation.

3.2 Fine Grained Analysis

In this section we propose a systematic dependence analysis approach based on thread-level dependence graphs (TLDG). The purpose of TLDG is to capture cross-thread dependences. We first introduce TLDG and then describe the use of this graph for code generation.

Without loss of generality, we first 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 blocks on that region are identical or the region is executed by only one block. These assumptions are for simplicity purpose, i.e. previous frameworks are fully capable of dealing with such additional complexities, so the assumption can be easily eliminated without major adjustment to our framework.

The TLDG of a code segment may contain a number of separate graphs as some statements have no dependences on others. For a loop inside a kernel, the loop is fully unrolled

when building the TLDG. (As CUDA is a *fine-grained* SPMD-threaded programming model, most GPU kernels do not have extremely large loops inside.) For loops with unknown trip-counts, the approach handles the loop separately in a way similar to the approach described in the previous section.

3.2.1 Construction of TLDG

The TLDG is a directed graph constructed based on the data and control dependences in the CUDA code, with awareness of the semantics of the warp/block logical hierarchy and synchronizations. TLDG reflects the dynamic dependence pattern during the execution of a GPU block.

To generate the node set of TLDG, we focus only on those statements that access shared data (e.g. arrays) from different threads in the warp or block. The first step of TLDG construction is breaking the statements into load/store references. We then divide the GPU code into Data Reference Units (DRU), each containing exactly 1 reference to shared data. Such DRUs will be the basic execution and scheduling unit in further transformations. Each DRU maps to a node, and each node is marked by the array reference in its corresponding DRU. There is no designation on which DRU all other private computation statements should belong to, so they are attached arbitrarily to an adjacent node. Therefore the entire code of 1 GPU thread is partitioned into a list of nodes. We then repeat such list by the number of threads in the block, so that each dynamic instance of each DRU has its own node to be mapped on. Therefore the node set of TLDG is always formed by repeating its own “base” subset, and each node mapped to its dynamic instance of DRU.

Next task is to connect the nodes via directed edges, where each edge $n_1 \rightarrow n_2$ represents 1 of 2 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 same or different threads,

where the type of data dependence could be either true, anti or output.

Note that nodes constructed from the same DRU are always executed simultaneously on GPU; and since there are no loops in the code, dependence edge can not point from a later DRU to an earlier DRU. Thus if we layout the nodes into a matrix, with the thread id increasing along the horizontal direction and the time stamp of each DRU being executed increasing along the vertical direction, then there should never exist edges pointing upwards.

Such static dependence analysis is done for each DRU against all DRUs after it in the GPU timeline. Since most array indices fall into the category of compile time known values, static analysis is capable of handling the common cases with moderate overhead. Detecting dependences resulting from dynamic array indices is also doable by marking all potentially overlapping accesses as dependent access. Such extension might introduce unnecessary edges into TLDG, but the simplicity of this solution makes it still worthwhile.

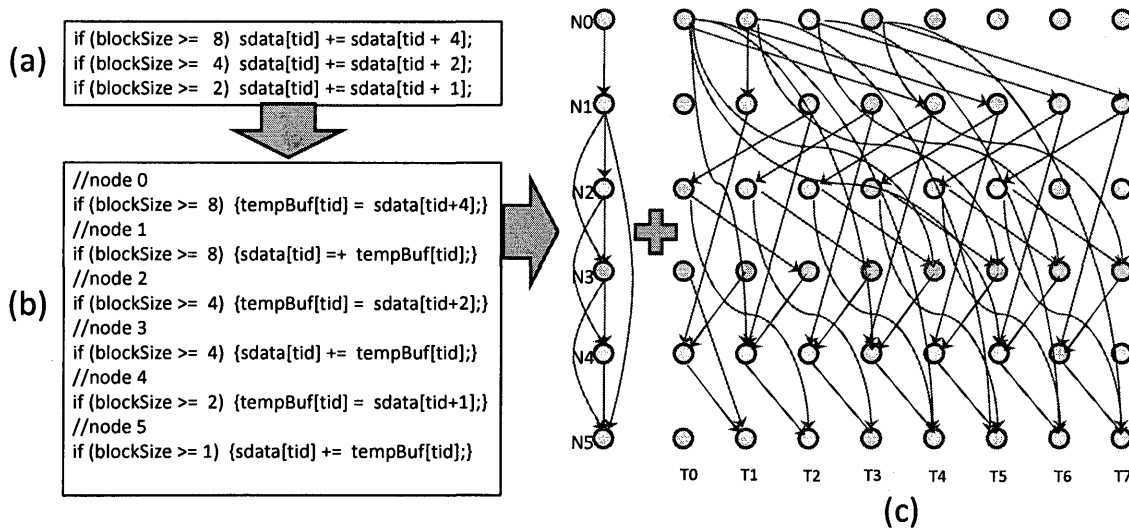


Figure 3.3: (a). The original statements in CUDA SDK source code. (b). Statements broken into references, each forming a DRU. (c). The intra-thread and inter-thread edges of the TLDG constructed from (b).

Figure 3.3 (c) shows the TLDG constructed from the last 3 unrolled statements CUDA SDK reduction code in figure 3.3 (b), where each DRU consists only 1 array reference

extracted from the original source code in figure 3.3 a. Nodes from the same thread are aligned along the vertical direction. For easier comprehension, the nodes are arranged in the aforementioned pattern such that along the top-down direction, they reflect the actual timeline of the original GPU execution of each block, and along the left-to-right direction, the index of all the homogeneous threads is increasing. A natural result of such arrangement is that all dependences in the graph are pointing from a higher positioned node to a lower one, e.g. there are no cycles in the graph. Therefore the blocks connected by dependence edges essentially form a partially ordered set, and the CPU translation of this code region is equivalent to the serialization of this set, which can be further reduced to finding one of the topologically sorted sequences of nodes that yields best CPU performance.

Chapter 4

Solutions

This chapter first presents several low-overhead approach to solve the correctness issue in the translation of implicit synchronizations. Then we present a TLDG-based approach that is capable of dealing with both implicit and explicit synchronizations, and discuss further optimization techniques of this approach, e.g. redundancy removal and code size reduction.

4.1 Coarse-Grained Solution

This section presents two solutions for handling implicit synchronizations. The first is based on the statement-level dependence analysis revealed in Section 3.1. The second is based on the simple extension described in Section 3.2, with the correctness issue on thread-dependent conditional branches addressed. The second solution is developed as the baseline for efficiency comparison.

4.1.1 Solution 1: A Dependence-Based Splitting-Oriented Approach

The first solution to implicit synchronizations is based directly on the SPMD-Translation Dependence Theorem. It consists of six steps to be conducted by compilers.

- Step 1: Apply the basic SPMD-translation to obtain thread loops for each code segment bounded by explicit synchronizations. Let LS represent the set of thread loops.

- Step 2: Extract a loop L from LS , compute the dependence sign vector from every statement (S) in L to all other statements in L that have a sequence number greater than that of S . Statements that access only thread-local data do not need to be considered in this step.
- Step 3: Based on the vectors, the dependences are classified into four sets: the intra-thread set I , inter-thread but benign set B , inter-thread but reversible set R , and inter-thread critical set K . Let d represent a data dependence and v be its dependence sign vector. The classification rules are as follows: $v \in I$ if $v(1) == 0$; $v \in B$ if v satisfies either condition 2 or 3 in the SPMD-Translation Dependence Theorem; $v \in R$ if the dependence can turn into a benign dependence when the index of the thread loop gets reversed; K consists of all other data dependences.
- Step 4: If $R == K == \phi$, the compilation is correct; go to Step 6.
- Step 5: Use the algorithm in Figure 4.1 to replace L with a sequence of loops; each loop has $(warpSize - 1)$ iterations and executes sequentially.
- Step 6: If $LS \neq \phi$, go to Step 2.

The algorithm used in Step 4 is based on two insights. First, as Rule 2 reveals, the loop form of a kernel is correct if it has only downward dependences, its loop index spans from 0 to $(warpSize - 1)$, and it runs sequentially. Second, let loop L be the loop form of a kernel and L contains only upward dependences. Let \hat{L} be a reverse form of L —that is, it has the same loop body as L does but its loop index takes a reverse order. Then, the sequential execution of \hat{L} must be correct because reversing loop index turns all upward dependences into downward dependences.

With the two insights, we explain the algorithm in Figure 4.1 based on an example shown in Figure 4.2. For simplicity of explanation, we first assume that there are no condition branches or loops in the kernel. The algorithm uses two sets, S_d and S_u to track statements that have been visited. At the beginning, the two sets are empty. Given a CFG, the

```

// SK: set of statements involved in critical dependences
// SB: set of statements involved in benign dependences
// SR: set of statements involved in reversible dependences
Sa = Sd =  $\phi$ ;
while (s = nextStatement()){ // in order of sequential numbers
  if (s $\in$  SK || (s $\in$  SB && s $\in$  SR)){
    createLoop_asc(Sa); // with ascending loop index
    createLoop_des(Sd); //with descending loop index
    createLoop_asc(s);
    Sa = Sd =  $\phi$ ;
  }
  else if (s $\in$  SR)
    Sd.add(s);
  else
    Sa.add(s);
}
createLoop_asc(Sa); // handle the final remaining statements if any
createLoop_des(Sd); // handle the final remaining statements if any

```

Figure 4.1: Algorithm for step 5 in Solution 1.

algorithm traverses the graph in an order as follows: All back edges are ignore; a node is not visited until all its predecessors have been visited.

During the traverse, if a statement s involves downward dependence only, it is put into S_d ; if involving upward dependence only, it is put into S_u . If it involves both types of dependence, the algorithm generates a loop for the current S_d (with an increasing loop index), a loop for the current S_u (with an decreasing loop index), and then a loop for s itself (with an increasing loop index; decreasing works too). This code generation ensures that the dependences of the original kernel can be observed. An example is the statement S_5 in Figure 4.2. After the code generation, both S_d and S_u are reset to empty.

The fifth step deserves some further explanations. It tries to fix dependence violations caused by the basic SPMD-translation. Its basic strategy is to split a problematic loop at some critical implicit synchronization points. These points are those statements involved in dependences belonging to either K or both B and R . In both cases, simple loop reversal is insufficient to fix the dependence violations. It uses set S_a to record statements that involve no inter-thread dependences or only benign dependences, and uses set S_d for those

involving inter-thread reversible dependences. At a splitting point, it creates a thread loop with an ascending index to enclose all statements in S_a , and a loop with a descending index to enclose all statements in S_d , and then puts the current statement into a single loop (which is likely to be unrolled in later optimizations). Both S_a and S_d are then set to empty. Figure 4.1 illustrates the algorithm.

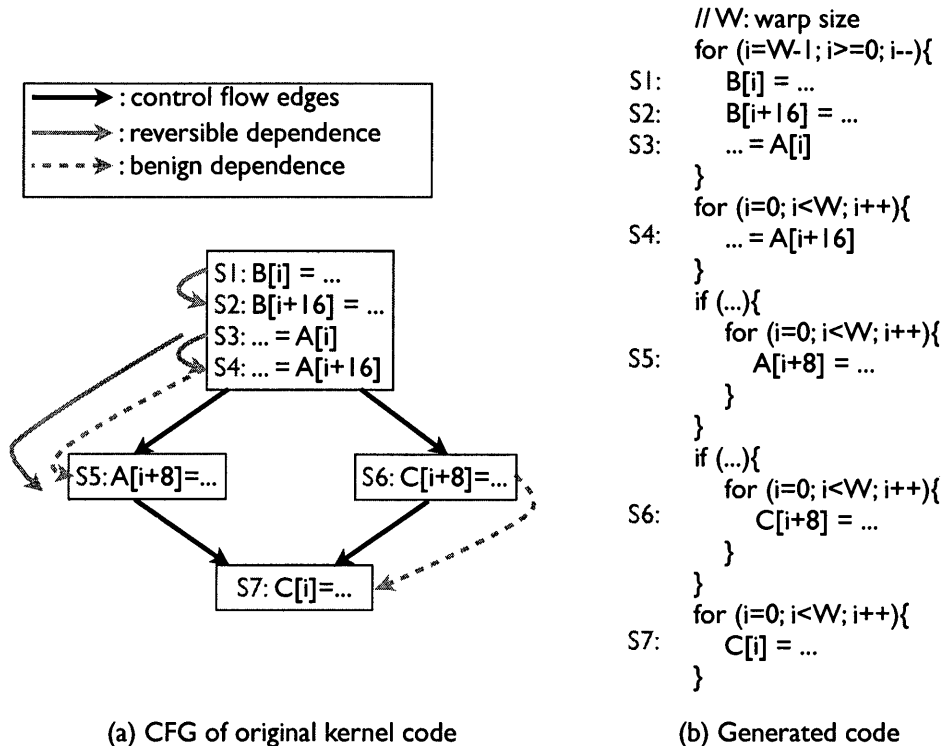


Figure 4.2: An example for Solution 1.

Control Dependences Certain constructs (e.g., if-else and loops) cause control dependences. We first briefly explain the treatment to constructs with conditional branches. If the CFG contains branches as exemplified by $S5$ and $S6$ in Figure 4.2, statements in a branch are treated similarly as the other statements, except that each of them are appended with a condition check at the front (e.g., $S5$ becomes “if (...) $A[i+8]=...$ ”). The condition to check is the boolean value checked in their enclosing “if” condition. Turning the statements into predicated statements creates much flexibility for code generation. Some bookkeeping is

needed if the condition is subject to change in the conditional branch. Condition hoisting is then used to refine the generated program (e.g., “for () { if (b) A[i+8]=...; if (b) A[i]=...;}” turns into “if (b) { for () { A[i+8]=...; A[i]=...;}}”).

For loops, no special treatment is necessary if their bounds are thread-independent or the loops contain no statement that involves an inter-thread critical dependence. Otherwise, some bookkeeping and code replication are needed as illustrated in Figure 4.3. In the example, there is a critical dependence between the first statement and the “if” condition. A complexity is that in the execution by a GPU warp, due to the SIMD mode, once a thread fails the “if” check, it won’t check that condition again. The introduction of the assistant array, `_cnt[]`, is to maintain such a property.

The code generation involves some necessary variable renaming (e.g., “i” becomes “iArr” in Figure 4.3) similar to the practice of prior SPMD-translations [13, 12].

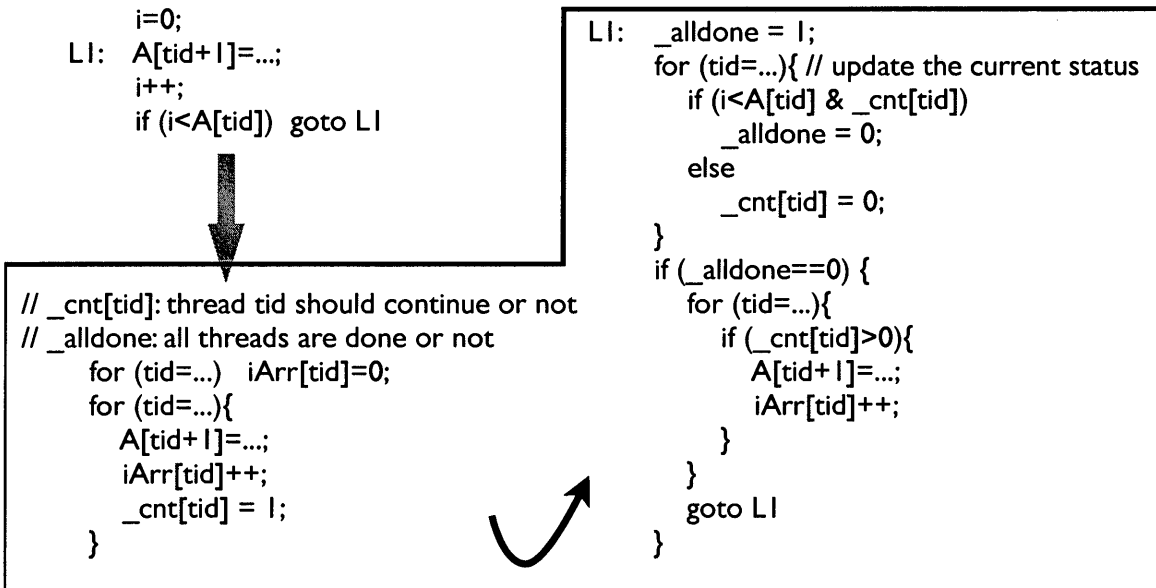


Figure 4.3: Illustration of translating a GPU loop with thread-dependent critical implicit synchronizations into CPU code.

4.1.2 Solution 2: A Merging-Oriented Approach

The second solution is based on the simple extension described in Section 3.1 with the correctness issue fixed. It treats all implicit synchronizations as explicit ones and uses the basic SPMD-translation for code generation. For thread-dependent implicit synchronizations, it uses the technique similar to the handling of control dependences in solution 1 (at the end of Section 4.1.1) to ensure correctness. The only difference is that it creates a loop for each statement. It then relies on the default loop fusion in compilers to reduce loop overhead. We develop this solution to serve as the baseline for our comparisons.

4.2 Fine-Grained Solution

In this section, we introduce a TLDG-based code generation scheme that are capable of dealing with both implicit and explicit synchronizations. Instead of treating a statement as a unit for dependence analysis, it distinguishes individual instances of a statement. As a fine-grained approach, it analyzes the relations among all instances of the statements, and exposes detailed dependence information, hence offering opportunities for exploiting both intra-thread parallelism and temporal locality.

```

while G not empty
  for each node N
    if N.inDeg == 0 //include nodes with no incoming edges
      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())

```

Figure 4.4: Pseudo code for round-based code generation

Our goal is to break the previous grid-level and block-level nested loop structure, and generate equivalent CPU grid level loop body directly from scratch. Specifically, we serialize the jobs in the same GPU block by generating sequential code from it, and utilize the task-

level parallelism among GPU blocks, thus reducing the fine-grained concurrency to fit CPU architecture. This means that 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 graph 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, thus generating a sequence of all DRUs in the GPU block that forms the body of grid-level loop in the CPU code. There's no need for a block-level loop since the sequence itself is the completely unrolled and reordered result of the original block-level loop.

An example algorithm framework is given in figure 4.4 that presents a round based code generation. The key idea is to partition the nodes into different groups and impose strict order among groups while maintaining full concurrency within each group. Therefore the DRUs in each group forms a round during 1 grid-level loop iteration. In each round, the algorithm pushes the set of all nodes with no incoming edges into the round group (roundQueue in figure 4.4), and deletes them along with their outgoing edges before proceeding to the next round, until the graph is empty. Eventually the round group will become a partition of the TLDG, and simply print out each of its elements in round order will produce a correct execution sequence of DRUs. This algorithm designates a round number to each DRU; as long as every block is executed after all the blocks with a lower round number, the source and sink relationships are preserved and the execution result is guaranteed correct. The successful detection and preservation of instance-level dependences effectively eliminated the need for a whole block synchronization, which is over conservative and strict. Such relaxation introduces an additional degree of freedom in the optimization space for GPU-CPU code compilation.

To minimize runtime overhead, the sequence is to be directly inserted into corresponding CPU functions. The code generation is a onetime process and the generated code can be reused provided that the workset to the CPU program will result in the same kernel configuration as the workset used to perform dependence analysis and generate this code.

Figure 4.5 (a) shows the content of a generated code.

This fine-grained code generation scheme has three advantages. First, the way a loop body is built is cache-friendly because it follows the dependence edges—data dependency means data reuse. Second, the resulting loops are all parallel loops, exposing more parallelism for exploitation (if wanted).

Last but not least, it exposes useless computations at the statement-instance-level. Traditional dead code elimination works on the statement level, which is insufficient for GPU programs. In GPU programs, sometimes even though all threads execute a statement, only some conduct useful work. An example is the final 6 lines of code in Figure 2.2 (b). The useless computations affect no GPU efficiency because of the massive parallelism of GPU; adding conditional statements to prevent them from happening may actually throttle GPU efficiency because extra condition checks are added into the critical path of every thread. But for CPU executions, such useless computations may hurt efficiency considerably. This instance-level solution can easily expose such useless instances and prevent them from getting into the generated CPU code.

A simple optimization technique is to build thread-dependent branching directions analysis into the code generation framework, thus eliminating the actual calculation of the branch variables at runtime. This technique also helps to reduce code size, resulting in slightly better cache performance.

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 so that the average distances between 2 adjacent references in the generated CPU code is minimized. Our experiments demonstrates that such reordering is beneficial to the overall performance on CPU.

4.2.1 Instance-level Redundancy Removal

The SIMD execution hides the overhead of redundant calculations. When encountering a divergent conditional branch within a warp, the GPU is forced to serialize the execution of the warp, iterating through all branch paths existing in the warp. Such divergences is a major source of performance bottlenecks [27], not to mention the condition calculation can also introduce extra overhead on the critical path. Therefore often unneeded GPU threads execute the same instructions and output result to the global memory just like other threads. In above translation, all these unnecessary calculations and/or conditional checks are kept in the generated CPU code.

Slightly extending the dependence analysis discussed above, a low cost data flow analysis for each individual node in the TLDG can be implemented with little extra overhead. The idea is to first identify upward exposed uses of the entries appearing in the code following TLDG code (the “valid entries”), then traversing the reversed TLDG only from those blocks that access the valid entries.

Pruning is particularly important to parallel reduction. 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, in the iteration process, no iterations after the first one actually utilizes more than half of all threads involved, but the redundant threads perform calculations just like the small portion of valid threads, creating huge waste of processor time that can only be hidden on GPU, and therefore considerable space for redundancy removal.

As depicted in figure 4.5, redundancy removal starts from a list of “useful” nodes in the TLDG and backtracks upwards to the top of the graph, marking all the useful nodes in the process. After redundancy removal, the number of lines of code generated for the reduction5 kernel with a configuration of block 256 threads is reduced from more than 3000 to around 500, resulting in a leap in the CPU program performance.

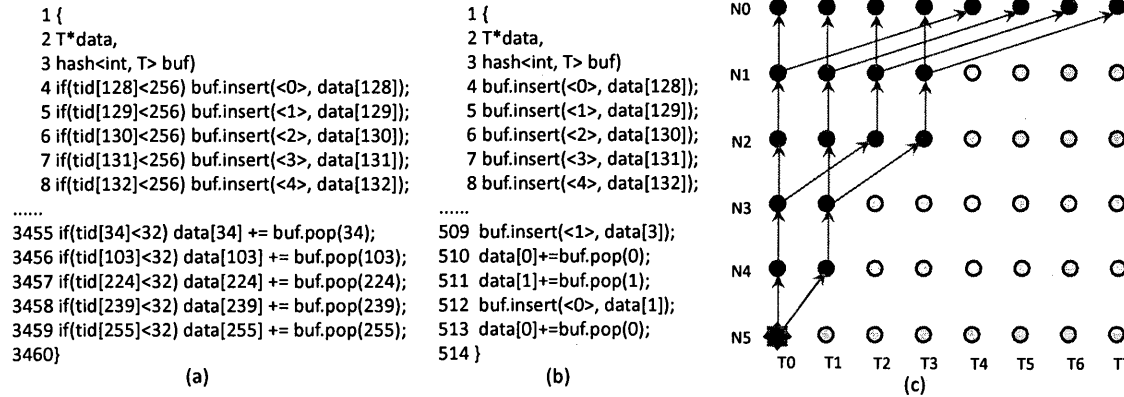


Figure 4.5: (a). The original hardcode without redundancy removal. (b). Pruned hardcode where all useless computations are removed. (c). The bottom-up redundancy removal process, start from the compiler identified useful final results. (marked black)

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 outperforms the original code with considerable speedup.

4.2.2 Discussion

An obvious benefit with such design is that there is no fundamental difference between the way TLDG treats implicit and explicit synchronization except on the number of GPU threads included into the graph. Therefore TLDG presents an universal solution that ensures both performance and flexibility, as well as space for further optimizations in all scenarios.

The iterative construction process of TLDG dictates that the graph always consists of a multitude of repeated basic homomorphous subgraphs, overlapping with each other. For example, the TLDG in figure 3.3 (a) can be further reduced into the subgraphs in figure 4.6 b, each repeated different in the horizontal direction. Such pattern extraction from the graph can lead to further reduction of the size of the CPU code. Instead of fully unrolling the block level loop into a linear function body, each of the subgraphs will form a

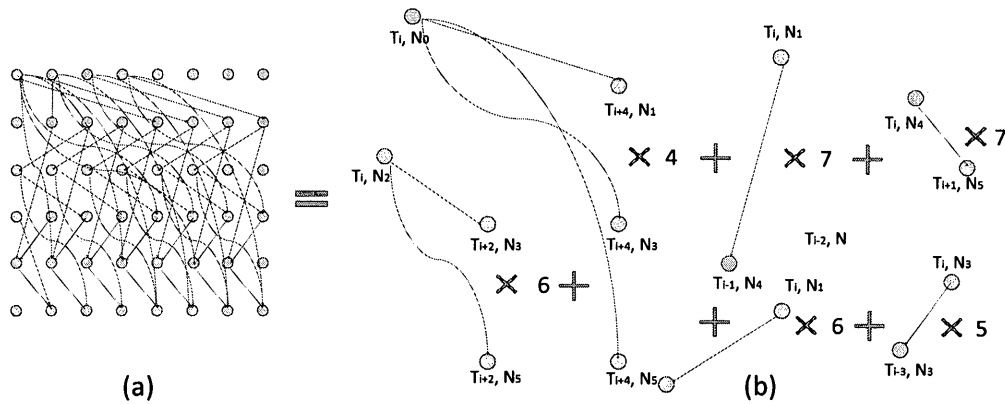


Figure 4.6: The original TLGD (upper left) broken down into 6 basic patterns, each of which retains its shape and orientation in the whole graph, only repeated on the horizontal direction.

loop body within the generated code with different trip counts. This additional dimension of flexibility provides the framework with the ability to leverage among multiple factors that might hurt the overall performance of the CPU program, including but not limited to instruction cache miss rate, loop overhead and basic block scope.

Chapter 5

Evaluation

In this chapter, we present the experiment results of both coarse-grained and fine-grained solutions. The coarse-grained solution puts the major emphasis in providing a low-overhead GPU-to-CPU translation that guarantees correctness, while the fine-grained solution focuses on utilizing the additional information and optimization space obtained from TLDG construction and redundancy removal to boost CPU code performance.

5.1 Coarse-Grained Solution

Our evaluation concentrates on two aspects: whether the proposed dependence-based solution can address the correctness issues in the basic SPMD-translation, and how efficient the produced code is.

5.1.1 Methodology

We use five benchmarks, listed in Table 5.1. They are selected because of their inclusion of non-trivial synchronizations, both explicit and implicit. Three of them, *Reduction*, *SortingNetworks*, and *TransposeNew* come from the NVIDIA CUDA SDK [3]. *CG* is a conjugate gradient application, originally from NPB [8] and later ported to CUDA as part of the H-PCGPU project [1]. *SGEMM* is a high performance linear algebra function developed by Volkov and Demmel [24].

All of the benchmarks contain a number of explicit synchronizations. The top three of them contain critical implicit synchronizations, while the other two do not. Including these two programs helps to examine the capability of the solutions in maintaining the basic efficiency of the program—that is, whether they degrade the performance of the part of code that contains no critical implicit synchronizations.

Table 5.1: Benchmarks

Program	Source	Description
CG	[1]	conjugate gradient
Reduction	[3]	parallel reduction
SortingNetworks	[3]	bitonic sort & odd-even merge sort
SGEMM	[24]	combined matrix matrix operations
TransposeNew	[3]	matrix transpose

To test the performance on different platforms, we run our experiments on two types of machines and through two compilers. One machine is a quad-core Intel Xeon E5640 machine. The other is a dual-socket dual-core AMD Opteron 2216 machine in the National Center for Supercomputing Applications. We call these machines the *Intel* and *AMD* machines respectively. Both machines run Linux (2.6.33 and 2.6.32). The Intel machine has GCC 4.1.2 and the AMD machine has Intel ICC 11.1 installed. All compilations use the highest optimization levels supported by the compilers.

5.1.2 Experimental Results

For each benchmark, we create three versions:

- *Basic Version*: This version is the result from the basic SPMD-translation in MCUDA [13]. MCUDA has limitations in handling some language-level features, for which, manual modifications are conducted.

- *Merging Version*: This version is the result from the merging-oriented solution described in Section 4.1.2. It is based on a straightforward extension to the basic SPMD-translation, but with issues on thread-dependent synchronizations addressed.
- *Splitting Version*: This version is the result from the dependence-based splitting-oriented solution described in Section 4.1.1.

Correctness The correctness of the three versions are as expected. For the three programs containing critical implicit synchronizations, some testing inputs cause the basic version to produce erroneous results. However, all testing results of the merging and splitting versions are correct. Manual code analysis confirms that in both versions, the errors on the critical implicit synchronizations in the basic version are fixed. All three versions output correct results on *TransposeNew* and *Sgemv* as they contain no critical implicit synchronizations.

Efficiency Figure 5.1 compares the performance of the three versions on the Intel machine when GCC is the compiler. Figure 5.2 shows the comparison on the AMD machine when ICC is used.

For the first three programs, it is important to note that the performance of the basic version is just for reference as they are erroneous. Because they give no treatment to implicit synchronizations, their code is the simplest and their executions finish the earliest. For these three programs, the performance comparison between the merging and splitting versions is more meaningful as both produce correct results.

For these three programs, the splitting version runs considerably faster than the merging version on the Intel machine especially on reduction and sortNet. The main reason is that the merging-oriented approach creates many small loops, and the loop overhead causes significant performance influence. The splitting-oriented approach, on the other hand, creates loops only when necessary based on the dependence analysis.

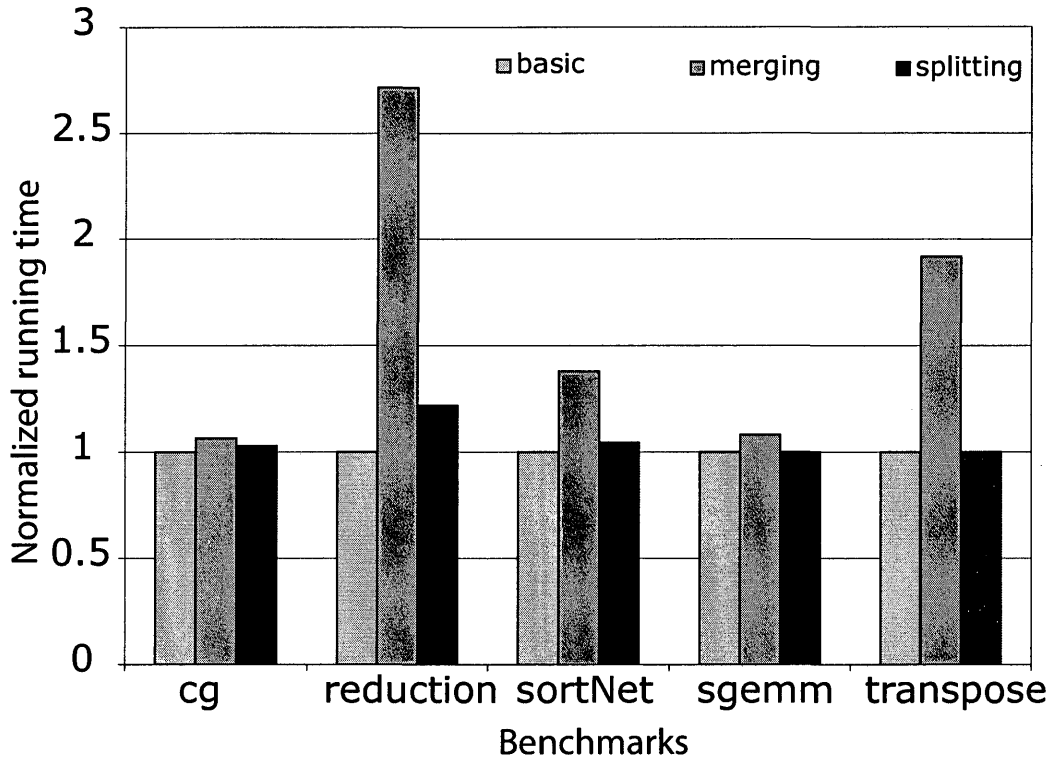


Figure 5.1: Running times on the Intel machine, normalized to the execution times of the (erroneous) basic SPMD-translation results. (“sortNet” is SortingNetworks in short)

The case of CG is trivially different in that the merge based version for this benchmark is created using more aggressive strategy, resulting in larger loop bodies and lower loop overhead due to implementation limitations. Nevertheless the splitting version is still faster than merging version, hence consistent with previous analysis.

As GCC has limited loop fusion functionality, it cannot remove overhead effectively. Because of that, we apply the commercial compiler, ICC, to the programs and run the same experiments on the AMD machine. As Figure 5.2 shows, the overhead of the merging version becomes smaller than on the Intel machine with GCC used, but is still substantial compared with the splitting version.

For the remaining two programs, all three versions are comparable as they are all correct. The splitting version shows similar performance as the basic version, indicating the capability of the dependence-based solution for maintaining the basic efficiency of the pro-

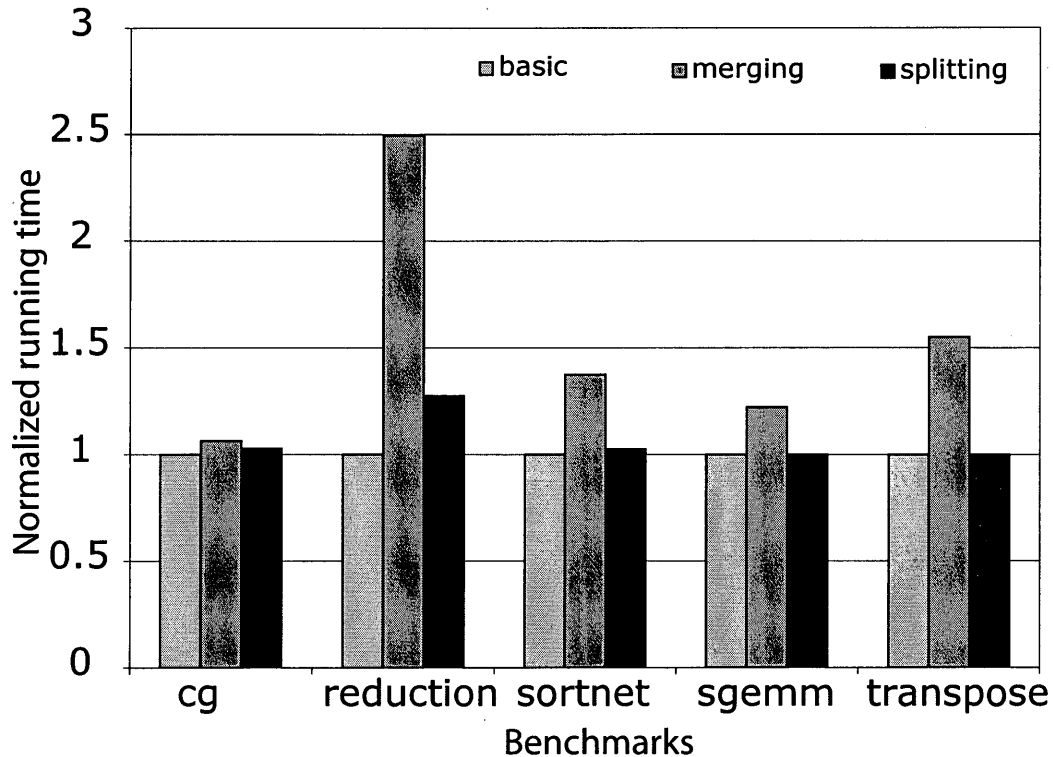


Figure 5.2: Running times on the AMD machine, normalized to the execution times of the (erroneous) basic SPMD-translation results. (“sortNet” is SortingNetworks in short)

grams. The merging version still causes considerable overhead because of the many, small loops created.

Overall, the dependence-based splitting-oriented approach demonstrates the promise to serve as an effective solution to the correctness issue of the basic SPMD-translation. It is able to correct the compilation error with the basic efficiency of the compilation results maintained.

5.2 TLDG-Based Solution

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. While our TLDG-based translation also

treats implicit synchronizations during the process, our evaluation mainly focuses on the improvement of program performance.

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 supported.

5.2.1 Versions

For each benchmark, we create 4 versions to compare the effect of our optimizations.

- *Baseline*: This version is generated by MCUDA, currently the best known source-level SPMD-translation tool available to the public. We introduced slight manual modifications in the program to make the different versions' results more comparable. Note that the translation results from this version might not be correct.
- *Merged Version*: With a simple extension to the MCUDA approach we can address the correctness pitfall, at the cost of large amount of small loops and increased loop overhead. This version is generated by first identifying all synchronizations of both kinds within the program and then treating them in the same way, which means that the statements between every pair of implicit synchronizations will also become a separate loop. We then employ existing compiler loop fusion techniques on the generated code to produce more efficient program.
- *Split Version*: Statement-level dependence analysis can potentially discover critical implicit dependences and insert appropriate barriers (i.e., loop fission) just at the critical points. This approach may avoid the drawbacks of the merge version in creating too many small loops. In this experiment, we implement this statement-level approach through manual code analysis and generation. A comparison with this approach will show the benefits of the fine-grained analysis by the TLDG-based method.
- *TLDG-basic Version*: As discussed in section 3.2, this version is based on instance-level dependence analysis within the block using code generation. The generated code

does not maintain the block-level loop structure like all 3 version above. Instead, it's in the form of a completely unrolled large linear function body repeated in the warp-level loop. All condition calculations are preserved in this version

- *TLDG-opt Version*: Based on TLDG-basic version, multiple optimization techniques are applied to obtain smaller code size and better memory access pattern, for example redundancy removal and reordering. Compile time condition calculations are built into code generation.

5.2.2 Experiment Results

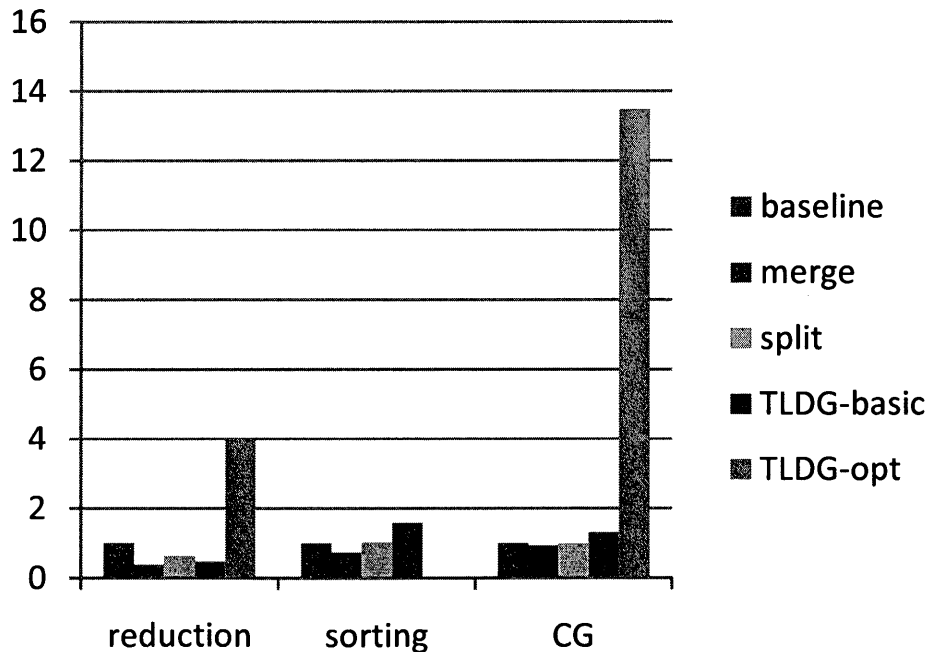


Figure 5.3: Relative performance compared to (incorrect) baseline version.

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 2 reduction bodies on the *common* array.

In figure 5.3, we compare the performance of the 5 versions presented in the previous subsection, all normalized against the incorrect baseline version. The first thing to notice the merge version lags behind all other versions with considerable slowdown. This is because the merge version relies entirely on the automatic compiler loop fusion in GCC to alleviate the extra loop overhead brought by over-conservative treatment of implicit synchronizations. The split version always demonstrates similar (but not better) performance as the baseline, proving the effectiveness of statement-level dependence analysis and the moderate overhead of the synchronizations inserted based on the analysis.

The TLDG-based version outperforms all other versions significantly in 2 of the 3 benchmarks even without further redundancy removal. One reason for such advantage is the compiler being able to optimize the linear code in TLDG version as an extremely large basic block. Both the compiler and processor pipeline therefore would have sufficient space to re-schedule and fuse the instructions. The reference reordering within each round in the CPU code also helped maintaining a necessary degree of memory locality.

The one exception is `reduction`, where TLDG-based version showed worst performance among all versions. One factor that might contribute to this result 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 large number of instructions from other threads. To avoid introducing unnecessary variables renaming, we instead introduced a temporary buffer in the generated code to store the middle results of each DRU, as well as its own thread id to cope with the frequent condition calculations in the `reduction5` kernel. As shown in figure 4.5 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 design however, introduces some additional memory accesses in TLDG-based version when compared to all other versions. With only 2 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 large numbers of memory load and store

from the swapping process. Both cases provide sufficient optimization space in the CPU code for the compiler to take advantage of.

The fifth bar in each group shows the effect of redundancy removal. The speedup can be directly attributed to the downsized CPU code with all useless operations and condition calculations removed. Again on CG benchmark with larger kernel size and block size(512) our framework yielded better performance. Reduction also have dramatical speedup, making TLDG-based version the most efficient even for this benchmark and clearly indicating the superiority of the instance-level dependence analysis approach. The shrink of code size from redundancy removal is also significant, namely 6.8x and 8x for reduction and CG, respectively.

Chapter 6

Related Work and Conclusion

6.1 Related Work

A number of previous works have aimed at automatic compilation of GPU program onto CPU. MCUDA [13, 12] and Ocelot [11]) both use an iterative execution framework based on the original GPU code structure to take advantage of its data and logical regularities. However, neither addresses the implicit synchronization pitfall. Given the large number of programs using this intrinsic, the correctness concern is unavoidable. Furthermore, the cost of maintaining much of the original threading structure in CPU is excessively strict constraints and waste of CPU time on useless operations.

NVIDIA provided a native emulation tool for running CUDA programs on CPU focuses on easing the debugging on GPU rather than improving performance [2]. Under emulation mode, the programmer needs to manually insert macros to judge the current device at runtime, and insert `__syncthreads()` when it finds itself running on CPU. Although CUDA emulator provides the capability to run GPU program on CPU, its pure manual usage dictates it unsuitable for general GPU-to-CPU compilation. A similar case lies in OpenCL. While it provides implicit synchronizations to the programmer, 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. [4]

There have been many studies trying to ease GPU programming. A common approach

is pragma guided semi-automatic OpenMP to CUDA compilation. [7, 18]. Others have proposed extensions to CUDA or OpenCL(e.g. [20]). Dynamically optimization of GPU executions through either software (e.g. [16, 10, 26, 9, 21, 27]) or hardware (e.g. [19, 23, 14]) techniques is also a well-studied area during recent year. However, to our best knowledge, there are no existing studies focusing on the efficient compilation of synchronizations on GPU; neither has instance level redundant work elimination on GPU been discussed in any previous works.

6.2 Conclusion

GPU-to-CPU translation plays a central role in the design of synergistic heterogeneous systems. Previous translation schemes failed to take into consideration the difference between GPU and CPU in programming model, execution model and underlying performance assumptions, therefore their translation result are often incorrect or inefficient. The problem lies in the understanding of the hierarchical logical structure of GPU program, the semantics of GPU specific intrinsics, and the constraints they impose on the translation.

This thesis first presents an SPMD-translation dependence theorem, and then propose two novel dependence analysis methods to reveal the impact of data dependences on the correctness and performance of GPU-to-CPU translation. The coarse-grained reference level dependence analysis demonstrates the relations between data dependences and the correctness of SPMD-translation regarding implicit synchronizations, while the TLDG-based instance-level dependence analysis captures fine-grained data and control dependence in the program, and uses the information to analyze the performance penalties resulting from naive translation of explicit and implicit synchronizations used in previous frameworks.

The second half of this thesis introduces systematic solutions for fixing the correctness and performance issues in current SPMD-translations. We propose several extensions to the current translation schemes to guarantee the correctness. Then we present a TLDG-based framework to use the fine-grained dependence analysis results for code generation and

useless computation elimination. Experiments show that the coarse-grained dependence-based extensions solve the problem effectively, with correct and efficient code produced for all tested benchmarks, while the TLDG-based fine-grained solution can further improve the performance of translated programs significantly.

On the high level, this work, for the first time, systematically examines the complexities that device-specific synchronizations create for heterogeneous computing. The extraction of the dependence information and the resulting extra flexibility may benefit practices beyond CUDA-to-CPU compilation.

Bibliography

- [1] Hpcgpu project. <http://hpcgpu.codeplex.com/>.
- [2] NVIDIA CUDA Programming Guide. <http://developer.download.nvidia.com>.
- [3] NVIDIA CUDA SDK. developer.nvidia.com/cuda-cc-sdk-code-samples.
- [4] OpenCL. <http://www.khronos.org/opencv/>.
- [5] R. ALLEN AND K. KENNEDY. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, 2001.
- [6] J. AUERBACH, D. F. BACON, P. CHENG, AND R. RABBAH. Lime: a java-compatible and synthesizable language for heterogeneous architectures. In *OOPSLA*, 2010.
- [7] EDUARD AYGADE, ROSA M. BADIA, DANIEL CABRERA, ALEJANDRO DURAN, MARC GONZALEZ, FRANCISCO IGUAL, DANIEL JIMENEZ, JESUS LABARTA, XAVIER MARTORELL, RAFAEL MAYO, JOSEP M. PEREZ, AND ENRIQUE 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.
- [8] D. BAILEY, J. BARTON, T. LASINSKI, AND H. SIMON. The NAS parallel benchmarks. Technical Report 103863, NASA, July 1993.
- [9] M. M. BASKARAN, U. BONDHUGULA, S. KRISHNAMOORTHY, J. RAMANUJAM, A. ROUNTEV, AND P. SADAYAPPAN. A compiler framework for optimization of affine loop nests for GPGPUs. In *ICS'08: Proceedings of the 22nd Annual International Conference on Supercomputing*, pages 225–234, 2008.
- [10] S. CARRILLO, J. SIEGEL, AND X. LI. A control-structure splitting optimization for gpgpu. In *Proceedings of ACM Computing Frontiers*, 2009.
- [11] GREGORY DIAMOS, ANDREW KERR, SUDHAKAR YALAMANCHILI, AND NATHAN 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.
- [12] JOHN A. STRATTON ET. AL. Mcuda: An efficient implementation of cuda kernels for multi-core cpus. In *LCPC'08*, 2008.

- [13] JOHN A. STRATTON ET. AL. Efficient compilation of fine-grained simd-threaded programs for multicore cpus. In *CGO'10*, 2010.
- [14] 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.
- [15] M. HARRIS. High performance computing with CUDA. In *Tutorial in IEEE Super-Computing*, 2007.
- [16] AMIR HORMATI, MEHRZAD SAMADI, MARK WOH, TREVOR MUDGE, AND SCOTT MAHLKE. Sponge: Portable stream programming on graphics engines. In *ASPLOS'11*, 2011.
- [17] 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.
- [18] SEYONG LEE, SEUNG-JAI MIN, AND RUDOLF EIGENMANN. Openmp to gpgpu: a compiler framework for automatic translation and optimization. In *PPOPP'09*, pages 101–110, 2009.
- [19] J. MENG, D. TARJAN, AND K. SKADRON. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *ISCA'10*, 2010.
- [20] STEUWER MICHEL, KEGEL PHILIPP, AND GORLATCH SERGEI. Skelcl - a portable skeleton library for high-level gpu programming. In *IPDPS'11*, 2011.
- [21] 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.
- [22] GUANGMING TAN, ZIYU GUO, MINGYU CHEN, AND DAN MENG. Single-particle 3d reconstruction from cryo-electron microscopy images on gpu. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 380–389, New York, NY, USA, 2009. ACM.
- [23] D. TARJAN, J. MENG, AND K. SKADRON. Increasing memory latency tolerance for simd cores. In *SC'09*, 2009.
- [24] V. VOLKOV AND J DEMMEL. Benchmarking gpus to tune dense linear algebra. In *2008 ACM/IEEE Conference on Supercomputing*, 2008.
- [25] M. WOLFE. Openmp on accelerators. In *Position paper*, 2010.
- [26] Y. YANG, P. XIANG, J. KONG, AND H. ZHOU. A gpgpu compiler for memory optimization and parallelism management. In *PLDI*, 2010.

- [27] EDDY ZHENG ZHANG, YUNLIAN JIANG, ZIYU GUO, KAI TIAN, AND XIPENG SHEN.
On-the-fly elimination of dynamic irregularities for gpu computing. In *ASPLOS'11*,
2011.

VITA

Ziyu Guo

Ziyu Guo received his Bachelor of Engineering degree from Harbin Institute of Technology in 2006, and Master of Engineering degree from the Graduate University of Chinese Academy of Sciences in 2009, in Computer Science and Computer Engineering respectively. He has been a Master student in the Department of Computer Science at the College of William and Mary since 2009. His research interests lie in compiler technology, workload characterization, GPU computing, parallel computing and computer architecture.