Correctly Treating Synchronizations in Compiling Fine-Grained SPMD-Threaded Programs for CPU

Ziyu Guo Eddy Z. Zhang Xipeng Shen
Computer Science Department
The College of William and Mary, Williamsburg, VA, USA 23187
{guoziyu,eddy,xshen}@cs.wm.edu

***Abstract*—Automatic compilation for multiple types of devices is important, especially given the current trends towards heterogeneous computing. This paper concentrates on some issues in compiling fine-grained SPMD-threaded code (e.g., GPU CUDA code) for multicore CPUs. It points out some correctness pitfalls in existing techniques, particularly in their treatment to implicit synchronizations. It then describes a systematic dependence analysis specially designed for handling implicit synchronizations in SPMD-threaded programs. By unveiling the relations between inter-thread data dependences and correct treatment to synchronizations, it presents a dependence-based solution to the problem. Experiments demonstrate that the proposed techniques can resolve the correctness issues in existing compilation techniques, and help compilers produce correct and efficient translation results.**

***Index Terms*—GPU; CUDA; GPU-to-CPU Translation; Implicit Synchronizations; Dependence Analysis; SPMD-Translation**

I. Introduction

With the rapid adoption of accelerators (e.g., GPGPU) in mainstream computing, heterogeneous systems are becoming increasingly popular. Programming such systems is a challenge. Many devices have their own programming models and hardware artifacts. 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.

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 [5]), libraries (e.g., OpenCL [3]), and cross-device compilers (e.g., CUDA Fortran compiler [25], O_2G [16],MCUDA [21], [22],Ocelot [10]).

A common challenge to virtually all these efforts is how to treat synchronizations in a program. Different types of devices often entail different restrictions on inter-thread relations. Except for some new streaming languages (e.g., Lime), device-specific synchronization schemes (e.g., the implicit synchronization in CUDA described later) are allowed to be used by programmers in their applications. Uses of such schemes help efficiency, but also cause complexities to cross-device code translations. Despite the importance of the issue for compilation correctness, current understanding to it remains preliminary.

In this paper, 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 point out a correctness pitfall current SPMD-translations are subject to. By revealing the relations between inter-thread data dependences and correct treatment to synchronizations, we present a solution to the problem and examine the influence on program performance.

An SPMD-translation takes a program written in a fine-grained SPMD-threaded programming model—such as CUDA [2]—as the base code, and generates programs suitable for multicore CPUs or other types of devices through compilation. 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. SPMD-translation simplifies coding for heterogeneous devices, and meanwhile, enables seamless collaboration of different devices (e.g., CPU and GPU) as tasks can be smoothly partitioned or migrated across devices. Recent efforts in this direction have yielded translators in both the source code level (e.g., MCUDA [21], [22]) and below (e.g., Ocelot [10]).

Our study uses CUDA as the fine-grained SPMD-threaded programming model for its broad adoption. There are two types of synchronizations in a CUDA program. The first is *implicit synchronization*, where despite the absence of synchronization statements, a group of threads must proceed in lockstep due to the artifacts of GPU devices. The second is explicit synchronization, enabled by some special statements.

The main focus of this paper is on implicit synchronizations. We show that the treatments to implicit synchronizations in current SPMD-translations are insufficient to guarantee the correctness of the produced programs (Section III). Through dependence analysis, we reveal the cause of the issue and the relations with various types of dependences in a program (Section IV and V).

Based on the findings, we then develop two solutions (Section VI). 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. This solution serves as a baseline approach for efficiency comparisons.

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

Overall, this work makes three main contributions:

- It initiates a systematic exploration to the implications of implicit synchronizations for SPMD-translations, and points out a correctness pitfall in current SPMD-translations.
- It reveals the relations between data dependences and pitfalls for handling synchronizations in SPMD-translations.
- It provides a systematic solution for addressing the correctness issue in existing SPMD-translators.

II. Background on CUDA and SPMD-Translation

This section provides some CUDA and SPMD-translation background that is closely relevant to the correctness issue uncovered in the next section.

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 is called **implicit synchronization**, 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 "__synchthreads()" for cross-warp synchronizations. That function works like a barrier, but only at the level of a thread block. In another word, no thread in a block can pass the barrier unless all threads in that block has reached the barrier. Such synchronizations are called **explicit synchronizations**. In CUDA, there is no ready-to-use 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, "__synchthreads()" 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 IV.

SPMD-Translation: The goal of SPMD-translation is to compile fine-grained SPMD-threaded programs to code acceptable by other types of devices. MCUDA [21], [22] 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 1 shows an example (with some simplifications for illustration purpose).

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 1) can run until all instructions before the synchronization point (e.g., the first loop in Figure 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

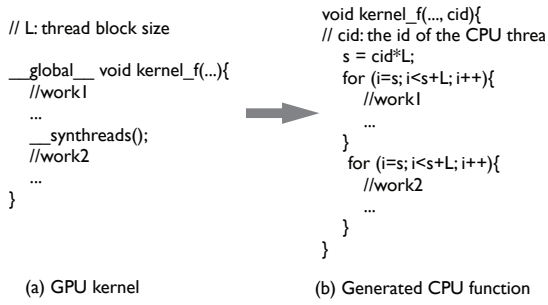

Fig. 1. Illustration of MCUDA compilation.

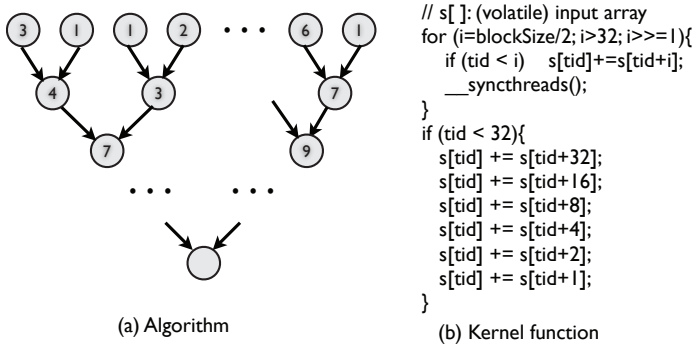

Fig. 2. Parallel reduction with implicit synchronizations used. (Assuming warp size=32, block size>= 64.)

ensures correct treatment to explicit synchronizations in a kernel through loop fission. As all existing SPMD-translation tools, MCUDA ignores implicit synchronizations in a kernel, which may cause erroneous translation results, as discussed next.

III. 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. We use source code statements rather than assembly instructions for the convenience of illustration.

Suppose that the "work1" in Figure 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 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+1] = 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 $B[warpSize]$) 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 [2]. 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 (a). Each middle level of the tree corresponds to one step in the reduction and computes the partial of the sum.

Figure 2 (b) shows a piece of code from the GPU kernel of the reduction program in CUDA SDK [2]. 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.

The six lines of code at the bottom of Figure 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 [14].

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. The issue jeopardizes their soundness and practical applicability.

IV. Insufficiency of Simple Extensions

A seemingly straightforward solution is to make current SPMD-translators treat implicit synchronizations the same as explicit synchronizations. This solution is insufficient for two reasons.

First, the simple extension may cause inefficient code being generated. Recall that MCUDA creates a loop for each code segment between two explicit synchronizations. As an implicit synchronization exists between every two instructions in a GPU kernel function, the simple solution may generate code consisting of a large number of loops, with each containing only one instruction in the loop body, entailing high loop overhead. Although loop fusion may help, its effectiveness is limited on such deeply fissioned code, as shown in Section VIII. Treating implicit synchronizations discriminatively

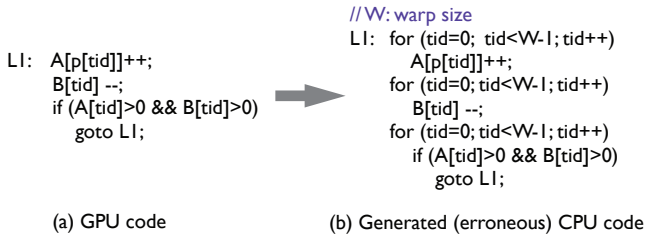

Fig. 3. Illustration of thread-dependent implicit synchronizations. Initially, $A = \{-1, 1\}$, $B = \{2, 2\}$, $P = \{1, 0\}$. The naively translated code (b) is erroneous.

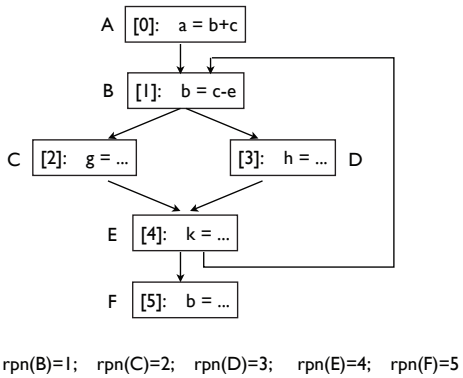

Fig. 4. Examples for the reverse postorder (rpn) of basic blocks and the sequence numbers (enclosed by "[]") of instructions.

would help, but that requires dependence analysis that handles complexities specific to implicit synchronizations, as elaborated later in this section.

The second issue is that the code generated by the simple extension may still be erroneous. As Section II mentions, MCUDA assumes that synchronization points are thread-independent, which holds for explicit synchronizations, but not for implicit synchronizations. Figure 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* $L1$ *and for only once*, the computing results are $A = \{1, 2\}$ and $B = \{1, 0\}$. But the execution of the CPU code produces results $A = \{1, 3\}$ and $B = \{0, 0\}$.

The reason for both issues of the simple extension exists in the differences between implicit and 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.

To accommodate these properties of implicit synchronizations, it is important to have a scheme to identify critical implicit synchronizations and generate code maintaining their 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 implicit synchronizations. We next present a systematic analysis of the relations, and then describe several derived solutions to the implicit synchronization problem.

V. Relations with Dependence

This section examines the relations between various dependencies and the correctness in compiling 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 dependence analysis is to first use the default (problematic) SPMD-translation scheme, as described in Section II, to derive a serial 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 II, 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.

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 4 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 4. 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 [4]. Roughly speaking, for a dependence from S1 to S2, the distance vector is the difference between the iteration vector of S2 and that of S1. 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 value of the corresponding loop index. For example, the dependence distance vector from S1 to S2 in the right graph of Figure 5 (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. By default, indexing of vector elements is 1-based.
- *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 5 (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 is not preserved by the basic SPMD-translation.
- $V(i : j)$. We use $V(i : j)$ to represent $V(i, i + 1, \cdots, j)$.

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. (As defined at the beginning of this section, C is for a GPU kernel containing no explicit synchronization, L is the serial loop produced from C by the basic SPMD-translation.)

Theorem 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 guaranteed to be preserved in L if and only if at least one of the following conditions holds:
(1) $v(1) == 0$;
(2) $v(1) > 0$ & all elements in $v(2 : |v|)$ are 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 5 exemplify that the correctness holds regardless the remaining elements of v.

For the second condition, because all elements in $v(2 : |v|)$ are zeros, 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 5 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 5 demonstrate that the correctness holds regardless the exact dependence directions between S_1 and S_2, while Figure 5 (h) shows a counter example.

So far we have proved the "if" part of the theorem. The "only if" part can be easily proved through an enumeration of all the cases of the sign vector that are not covered by the three conditions. It is elided in this paper.

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

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

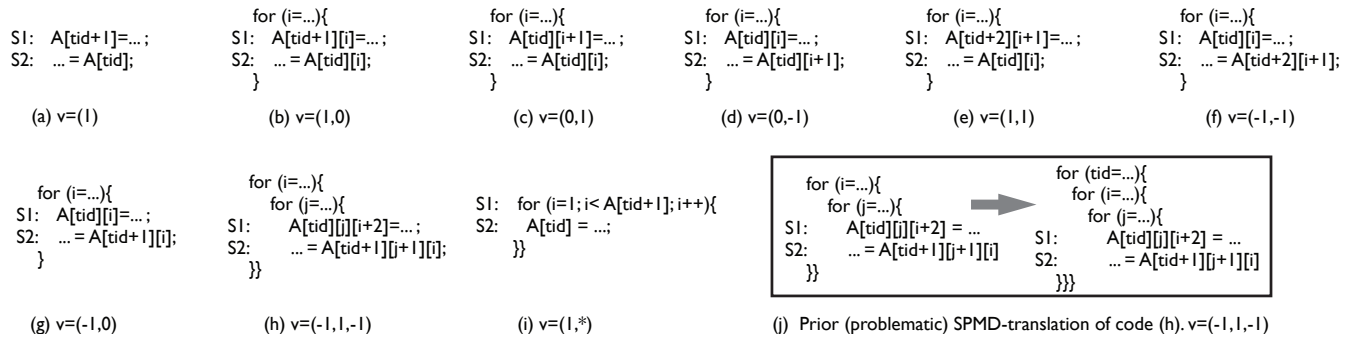

Fig. 5. Examples for demonstrating the SPMD-Translation Dependence Theorem. The code segments (a) to (i) are examples of GPU kernel code. The captions show the dependence sign vectors from $S1$ to $S2$ in their corresponding CPU code produced by the basic SPMD-translation, as illustrated by graph (j). 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.)

Figure 5 (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 VI 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 5 (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 would turn 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. Section VI will show how this implication can be systematically exploited in a modified SPMD-translation.

VI. Solutions

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

A. 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 6 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.

```
// 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 = ϕ;
while (s = nextStatement()){ // in order of sequential numbers
   if (s∈ SK || (s ∈ SB && s∈ SR){
      createLoop_asc(Sa); // with ascending loop index
      createLoop_des(Sd); //with descending loop index
      createLoop_asc(s);
      Sa = Sd = ϕ;
   }
   else if (s∈ 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
```

Fig. 6. Algorithm for step 5 in Solution 1.

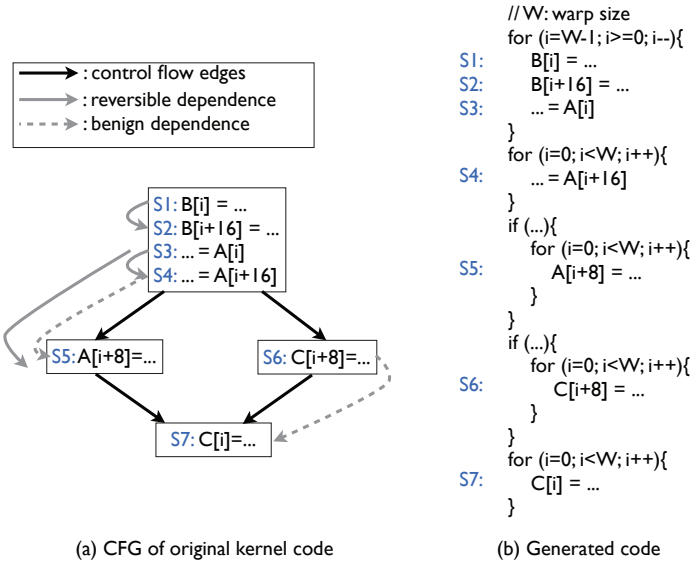

Fig. 7. An example for Solution 1.

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 6 outlines the algorithm.

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 7, statements in a branch are treated similarly as the other statements, except that each of them is turned into a predicated statement with the guarding condition derived from the original "if" statement enclosing them. 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 8. 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 8) similar to the practice of prior SPMD-translations [21], [22].

B. Solution 2: A Merging-Oriented Approach

The second solution is based on the simple extension described in Section IV 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 VI-A) 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.

VII. Discussions

Although the dependence theorem described in the previous section is general, its application in practice needs consideration of some actual complexities. For instance, if the code complexity makes it difficult to determine the dependence sign vector between two statements, the dependence should be considered as critical. In addition, CUDA supports some warp-level intrinsics (e.g., __any() and __all()). A warp-level barrier (i.e., loop fission) is typically needed after each invocation of such intrinsics in the generated code.

VIII. Evaluation

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.

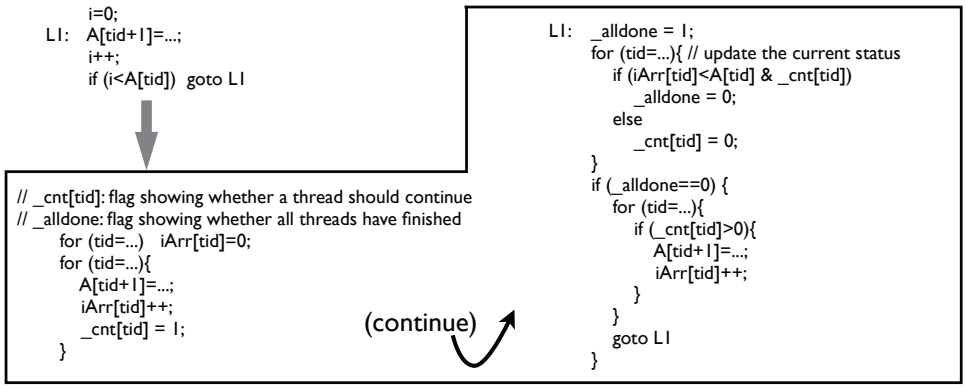

Fig. 8. Illustration of translation of a GPU loop with thread-dependent critical implicit synchronizations.

TABLE I
BENCHMARKS

Program	Source	Description
CG	[1]	conjugate gradient
Reduction	[2]	parallel reduction
SortingNetworks	[2]	bitonic sort & odd-even merge sort
SGEMM	[24]	combined matrix matrix operations
TransposeNew	[2]	matrix transpose

A. Methodology

We use five benchmarks, listed in Table I. 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 [2]. *CG* is a conjugate gradient application, originally from NPB [7] and later ported to CUDA as part of the HPCGPU 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.

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.

B. Experimental Results

For each benchmark, we create three versions. The first version is mainly through MCUDA [21]. The second and third versions are from the analyses and code generations described in Section VI. Although our application of the techniques is mainly manual, they can be automated through a compiler.

- *Basic Version:* This version is the result from the basic SPMD-translation in MCUDA [21]. 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 VI-B. 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 VI-A.

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. For example, when an array with 2^{24} '1's is used as input, the reduction produced 200704 rather than the correct answer 16777216.

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 *Sgemm* as they contain no critical implicit synchronizations.

Efficiency: Figure 9 compares the performance of the three versions on the Intel machine when GCC is the compiler. Figure 10 shows the comparison on the AMD machine when ICC is used. All these results record only the execution times of the kernels of interest.

For the first three programs, it is important to note that the performance of the basic versions 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 consid-

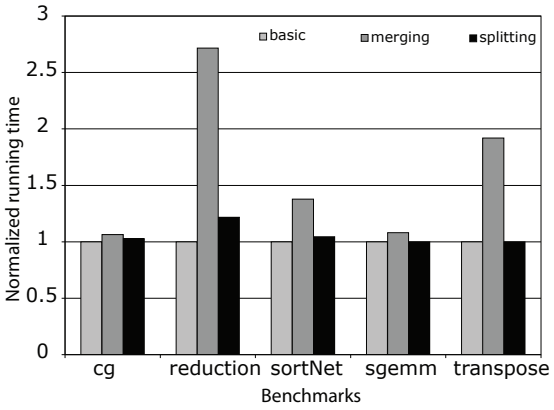

Fig. 9. Running times on the Intel machine, normalized to the execution times of the (erroneous) basic SPMD-translation results. ("sortNet" is SortingNetworks in short)

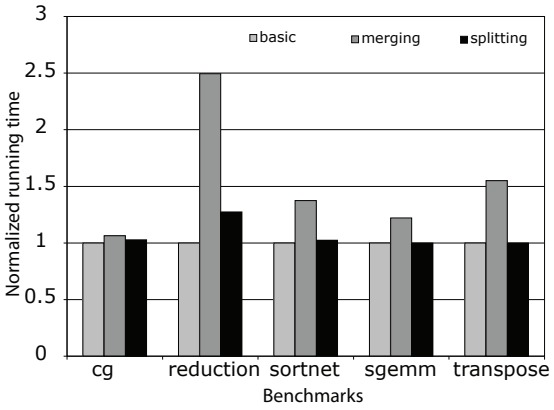

Fig. 10. Running times on the AMD machine, normalized to the execution times of the (erroneous) basic SPMD-translation results. ("sortNet" is SortingNetworks in short)

erably faster than the merging version on the Intel machine. 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. This difference in loop overhead has a large impact on the overall performance of *reduction* and *sortnet*. The small impact on *cg* is due to artificial reasons. Due to the complex code of *cg*, when creating the merging-oriented version, we did not break the kernel into as many loops as it should be. The performance of the merging-oriented version is hence much better than a rigorous implementation of that version.

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 10 shows, the overhead of the merging version becomes smaller than on the Intel machine when GCC is 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 programs. The merging version still causes considerable overhead because of the many, small loops created (recall that in this version, loops are created regardless whether the implicit synchronization is critical).

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.

IX. Related Work

Programming heterogeneous devices has drawn lots of recent attentions. Some researchers try to develop new languages (e.g., Lime in the Liquid Metal project [5]) to support multiple types of devices. Some attempt to build new libraries and compilation tools (e.g., OpenCL [3], MCUDA [21], [22], Ocelot [10]) to enable cross-device code generation from some existing (or slightly modified), broadly adopted programming models. But none of prior studies has systematically explored the issues related to device-specific synchronization schemes—such as the implicit synchronizations on GPU. Consequently, many of existing cross-device code generators are subject to correctness pitfalls. For instance, besides the discussions on MCUDA and Ocelot earlier in this paper, we have observed that OpenCL, the GPU programming model promoted by multiple industry companies, allows the use of implicit synchronizations but gives no specification on how they should be handled on different platforms. CUDA emulation addresses the issue by asking for programmers' annotations [2].

There have been many studies trying to improve GPU programming productivity by providing openMP to CUDA compilers [6], [11], [16], [25], or extensions to CUDA or OpenCL (e.g. [12], [20]). A few recent studies have concentrated on synergistic collaborations between CPU and GPU. Wu and others initiated a study of using GPU to perform runtime optimizations for CPU programs [26], while Zhang and others have demonstrated the potential of CPU-GPU pipelining for streamlining GPU computation on the fly [28]. Luk and others have exploited work partition between GPU and CPU through compiler and runtime support [17]. In addition, recent years have seen many efforts in optimizing GPU executions through either software (e.g. [8], [9], [15], [19], [27]–[29]) or hardware (e.g. [13], [18], [23]) approaches. We are not aware of previous studies on optimizing the treatment to implicit synchronizations.

X. Conclusion

In this paper, we present an SPMD-translation dependence theorem, and reveal the relations between data dependences and the correctness of SPMD-translation regarding implicit synchronizations. We introduce a systematic solution for fixing

the correctness issue in current SPMD-translations. Experiments show that the dependence-based techniques solve the problem effectively, with correct and efficient code produced for all tested benchmarks. On the high level, this work, for the first time, systematically examines the complexities that device-specific synchronizations create for heterogeneous computing. The insights may benefit practices beyond CUDA-to-CPU compilation.

ACKNOWLEDGMENT

We owe the anonymous reviewers our gratitude for their helpful suggestions on the paper. This research was enlightened by a question raised by Youfeng Wu at Intel. We thank the NCSA at UIUC for experimental devices. 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. http://www.nvidia.com/cuda.
[3] OpenCL. http://www.khronos.org/opencl/.
[4] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, 2001.
[5] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. Lime: a java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 2010.
[6] E. Ayguade, R. Badia, D. Cabrera, A. Duran, M. Gonzalez, F. Igual, D. Jimenez, X. Martorell, R. Mayo, J. Perez, and E. Quintana-Orti. A proposal to extend the openmp tasking model for heterogeneous architectures. In *Proceedings of International Workshop on OpenMP*, 2009.
[7] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report 103863, NASA, July 1993.
[8] 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.
[9] S. Carrillo, J. Siegel, and X. Li. A control-structure splitting optimization for gpgpu. In *Proceedings of ACM Computing Frontiers*, 2009.
[10] G. Diamos, A. Kerr, and M. Kesavan. Translating gpu binaries to tiered simd architectures with ocelot. 2009.
[11] D.Unat, X.Cai, and S. Baden. Mint: Realizing cuda performance in 3d stencil methods with annotated c. In *Proceedings of International Conference on Supercomputing*, 2011.
[12] J. Enmyren and C. Kessler. Skepu: A multi-backend skeleton programming library for multi-gpu systems. In *Proceedings 4th Int. Workshop on High-Level Parallel Programming and Applications*, 2010.
[13] 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.
[14] M. Harris. High performance computing with CUDA. In *Tutorial in IEEE SuperComputing*, 2007.
[15] A. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: portable stream programming on graphics engines. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.
[16] 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.
[17] C. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the International Symposium on Microarchitecture*, 2009.
[18] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *ISCA*, 2010.
[19] 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.
[20] M. Steuwer, P. Kegel, and S. Gorlatch. Skelcl a library for portable high-level programming on multi-gpu systems. In *Proceedings of International Parallel and Distribute Processing Symposium (IPDPS)*, 2011.
[21] J. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W. Hwu. Efficient compilation of fine-grained spmd-threaded programs for multicore cpus. In *CGO '10: Proceedings of the International Symposium on Code Generation and Optimization*, 2010.
[22] J. Stratton, S. Stone, and W. Hwu. Mcuda: An efficient implementation of cuda kernels for multi-core cpus. 2008.
[23] D. Tarjan, J. Meng, and K. Skadron. Increasing memory miss tolerance for simd cores. In *SC*, 2009.
[24] V. Volkov and J. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.
[25] M. Wolf. Openmp on accelerators. Position paper.
[26] B. Wu, E. Zhang, and X. Shen. Enhancing data locality for dynamic simulations through asynchronous data transformations and adaptive control. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2011.
[27] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A gpgpu compiler for memory optimization and parallelism management. In *PLDI*, 2010.
[28] E. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.
[29] 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.