Leveraging GPUs Using Cooperative Loop Speculation

MEHRZAD SAMADI, University of Michigan
AMIR HORMATI, Google Inc.
JANGHAENG LEE and SCOTT MAHLKE, University of Michigan

Graphics processing units, or GPUs, provide TFLOPs of additional performance potential in commodity computer systems that frequently go unused by most applications. Even with the emergence of languages such as CUDA and OpenCL, programming GPUs remains a difficult challenge for a variety of reasons, including the inherent algorithmic characteristics and data structure choices used by applications as well as the tedious performance optimization cycle that is necessary to achieve high performance. The goal of this work is to increase the applicability of GPUs beyond CUDA/OpenCL to implicitly data-parallel applications written in C/C++ using speculative parallelization. To achieve this goal, we propose *Paragon*: a static/dynamic compiler platform to speculatively run possibly data-parallel portions of sequential applications on the GPU while cooperating with the system CPU. For such loops, Paragon utilizes the GPU in an opportunistic way while orchestrating a cooperative relation between the CPU and GPU to reduce the overhead of miss-speculations. Paragon monitors the dependencies for the loops running speculatively on the GPU and nonspeculatively on the CPU using a lightweight distributed conflict detection designed specifically for GPUs, and transfers the execution to the CPU in case a conflict is detected. Paragon resumes the execution on the GPU after the CPU resolves the dependency. Our experiments show that Paragon achieves 4x on average and up to 30x speedup compared to unsafe CPU execution with four threads and 7x on average and up to 64x speedup versus sequential execution across a set of sequential but implicitly data-parallel applications.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Code generation, Compilers*

General Terms: Design, Performance

Additional Key Words and Phrases: Compiler, GPU, speculation, optimization

ACM Reference Format:
Mehrzad Samadi, Amir Hormati, Janghaeng Lee, and Scott Mahlke. 2014. Leveraging GPUs using cooperative loop speculation. ACM Trans. Architec. Code Optim. 11, 1, Article 3 (February 2014), 26 pages.
DOI: http://dx.doi.org/10.1145/2579617

1. INTRODUCTION

In recent years, multicore CPUs have become commonplace, as they are widely used not only for high-performance computing in servers but also in consumer devices such as laptops and mobile devices. Besides CPUs, GPUs have presented programmers with a different approach to parallel execution. Researchers have shown that for applications that fit the execution model of GPUs, in the optimistic case, speedups of 100–300x [NVIDIA 2010] and, in the pessimistic case, speedups of 2.5x [Lee et al. 2010] can be

This article extends an earlier version that appeared in the 5th Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU 2012), 2012.
Authors' addresses: Mehrzad Samadi (corresponding author), Janghaeng Lee, and Scott Mahlke, Computer Science and Engineering Department, University of Michigan, 2260 Hayward Street, Ann Arbor, MI 48109; email: mehrzads@umich.edu; Amir Hormati, Google Inc., 651 N 34th Street, Seattle, WA 98103.

DOI: http://dx.doi.org/10.1145/2579617

achieved between the most recent versions of GPUs compared to the latest multicore CPUs.

The main languages for developing applications for GPUs are CUDA and OpenCL. While they try to offer a more general-purpose way of programming GPUs, extracting high performance from GPUs is still a daunting challenge. Difficulty in extracting massive data-level parallelism, utilizing the nontraditional memory hierarchy, complicated thread scheduling and synchronization semantics, and lack of efficient handling of control instructions are the main complications that arise while porting applications to GPUs [Ryoo et al. 2008]. As a result of this complexity, the computational power of graphics engines is often underutilized or not used at all.

Although many researchers have proposed new ways to solve these problems [Brunie et al. 2012; Coutinho et al. 2011; Zhang et al. 2011; Xiao and chun Feng 2010], there is still no solution for an average programmer to target GPUs. To efficiently run a sequential or parallel (for small number of cores) C/C++ application on a GPU, there are two primary methods used by developers: manually redesigning the underlying algorithm of an application for GPUs to get rid of the memory and control bottlenecks, or using a compiler to perform automatic parallelization. In most cases, it is difficult to manually identify the bottlenecks and redesign an application for the massively data-parallel execution engines of GPUs. This solution is clearly not suitable for average programmers and is often expensive to apply due to the cost of reimplementing and redesigning large chunks of legacy applications. The second solution is to use compiler analysis to automatically extract enough data parallelism from an application to gain some performance benefit from the resulting code on the target GPU. In many cases, ambiguous memory dependencies or control flow divergences in a small number of threads can negatively affect thousands of other threads on a GPU. The main problem with this approach is that the compiler analyses used for automatic parallelization are usually too conservative and fragile, resulting in small or no performance gains on most commodity computer systems.

In this work, we take a different approach to this problem. Considering the amount of parallelism exposed by GPUs and their ubiquity in consumer devices, we propose cooperative speculative loop execution on GPUs and CPUs using *Paragon* for implicitly data-parallel programs written in C/C++. Paragon, using data-parallel speculation and distributed conflict detection engines carefully designed for cores in GPUs, enables programmers to transparently take advantage of GPUs for pieces of their applications that are possibly data parallel without manually changing the application or relying on complex compiler analyses, thus reducing the cost of migrating to GPUs. Further, the set of applications that can be mapped onto a GPU is broadened beyond loops that exclusively use arrays with affine indices. Paragon's use of cooperative execution between the GPU and CPU increases the performance of the overall system in the presence of conflicts since the CPU is not left idle while the GPU is speculatively running an application.

The idea of speculative loop execution is not a new one. Speculative parallelization has been extensively investigated in both hardware and software (see Section 7) in the context of multicore CPUs [Steffan et al. 2005; Kim et al. 2012; Volos et al. 2009; Harris et al. 2006; Mehrara et al. 2009; Oancea and Mycroft 2008; Tian et al. 2008]. However, speculation techniques for multicore CPUs are not designed to scale to thousands of active threads and deal with the complex memory hierarchy available on GPUs. Paragon's compilation and runtime system is the first system, that we are aware of, that explores the idea of cooperative speculation by leveraging GPUs and CPUs simultaneously while using lightweight and scalable conflict detection and recovery for large numbers of data-parallel threads. In Paragon, the CPU is used to execute parts of an application that are sequential, and both the GPU and CPU are utilized

for execution of possibly parallel for-loops. The GPU and CPU both start executing their version of a possibly parallel for-loop (sequential on the CPU, data parallel on the GPU). The GPU executes the for-loop assuming there is no data dependency between the iterations but monitors all the active threads for possible dependency violations. If a dependence violation is detected, the GPU waits for the execution of the dependency on the CPU and then resumes the remaining iterations. This approach puts otherwise idle GPUs to productive use, albeit at the cost of energy efficiency.

The Paragon compilation system is divided into two parts: *static compilation for speculation* and *cooperative execution management*. The static part mainly performs loop classification and generates CUDA code for the runtime system, which monitors the loops on the GPU for dependency violations. The execution management also performs lightweight one-time loop monitoring and decides which loops are more likely to benefit from executing on the GPU. These two phases together enable the execution of C/C++ loops with statistically improbable cross-iteration data dependencies on the GPU.

In summary, the main contributions of this work are:

—Static compilation and runtime systems for cooperative speculative execution on GPU/CPUs
—Lightweight runtime conflict detection on GPUs
—Low overhead rollback mechanism by using the concurrency between GPUs and CPUs

The rest of the article is organized as follows. In Section 2, the CUDA programming model and the basics of GPUs' architecture are discussed, as is the motivation behind Paragon. A brief overview of Paragon is discussed in Section 3. In Section 4, Paragon's compilation phases and its lightweight data-parallel speculative execution mechanism with distributed conflict detection on GPUs are explained. Section 5 discusses how cooperative execution works and explains advantages/disadvantages of using Paragon in different cooperative execution scenarios. Experiments are shown in Section 6. Finally, in Section 7, we discuss related works.

2. BACKGROUND AND MOTIVATION

2.1. CUDA Programming Model

The CUDA programming model is a multithreaded SIMD model that enables implementation of general-purpose programs on heterogeneous GPU/CPU systems. A CUDA program consists of a host code segment that contains the sequential sections of the program, which is run on the CPU, and a parallel code segment that is launched from the host onto one or more GPU devices. Host code can be multithreaded, and in this work, Paragon launches two threads on the CPU: one for managing GPU kernels and transferring data and the other one for performing computations. Recent generations of NVIDIA's GPUs, Fermi and Kepler, can support concurrent kernel execution, where different kernels of the same application context can execute on the GPU at the same time. Concurrent kernel execution allows programs that execute a number of small kernels to utilize the whole GPU. It is also possible to overlap data transfers between CPU and GPU and kernel execution.

The basic block of work in CUDA is a single *thread*. A group of threads executing the same code are combined together to form a *thread block* or simply a *block*. Threads within a thread block are synchronized together through a barrier operation (*__syncthreads*()). *Shared memory* is an on-chip memory shared only by threads within the same thread block. The last level of memory is *global memory*, which is an off-chip memory that is accessible to all threads of the kernel.

2.2. Motivation

Parallelizing an existing single-threaded application for a multicore system is often more challenging as it may not have been developed to be easily parallelized in the first place. It will be even harder to extract the fine-grained parallelism necessary for efficient use of many-core systems like GPUs with thousands of threads. Therefore, several automatic static parallelization techniques for GPUs have been proposed to exploit more parallelism [Han and Abdelrahman 2010; Baskaran et al. 2010; Wolfe 2010; Leung et al. 2009; Tarditi et al. 2006].

However, even the best static parallelization techniques cannot parallelize programs that contain irregular dependencies that manifest infrequently or statically unresolvable dependencies that may not manifest during runtime at all. Removing these dependencies speculatively will dramatically improve the parallelization possibilities. This work optimistically assumes that these programs can be executed in parallel on the GPU and relies on a runtime monitor to ensure that no dependency violation is produced.

Applications that are implicitly data parallel but at the same time difficult to parallelize often contain array index expressions that cannot be statically analyzed. We have identified three common types of loops that demonstrate this property: *nonlinear array access*, *indirect array access*, and *array access through pointers*.

Nonlinear array access. If a loop accesses an array with a nonlinear function of the loop's induction variables, it is hard to statically disambiguate the loop-carried dependencies. To illustrate, Figure 1(a) shows the `make_lattice()` function in the *milc* benchmark from SPEC2006. This function accesses the `lattice` array with the index `i`, which depends on the induction variables (`x, y, z, and t`) and the loop-independent variable `squaresize`. As shown in lines 4 to 8 of Figure 1(a), the index is calculated through modulo operation with loop-independent variables, which makes it difficult to disambiguate cross-iteration dependencies at the compile time. In fact, this loop may or may not have dependencies between iterations depending on `squaresize`.

Indirect array access. This type of access occurs when an array index is produced in runtime. For example, Figure 1(b) shows the code for forward elimination of a matrix in Compressed Sparse Row (CSR) format where suffix `L` denotes the array for the lower triangular matrix. Forward elimination is generally used as a part of the Gaussian elimination algorithm, which changes the matrix to a triangular form to solve the linear equations. CSR uses three arrays to store a sparse matrix: (1) a real array `a [1:nnz]` contains the nonzero elements of the matrix row by row, (2) an integer array `ja [1:nnz]` stores the column indices of the nonzero elements stored in `a`, and (3) an integer array `ia [1:n+1]` contains the indices to the beginning of each row in the arrays `a` and `ja`.

Like the previous example, a static compiler cannot determine whether these loops are parallelizable since the inner loop in Figure 1(b) accesses arrays using another array value as an index, which can be identified only at runtime. Since the inner loop is a sparse dot product of the `i`th row of array `a` and the dense vector `x`, runtime profiling will categorize this loop as a parallel loop.

Array access through pointers. This type of access also makes it difficult for static compilers to parallelize a loop. Figure 1(c) shows a function that simply adds two vectors taking pointers as parameters. If there is a possibility that the pointer `c` overlaps with either `a` or `b`, the loop cannot be parallelized. The conservative static compiler will give up parallelizing the loop if there is any chance of pointer aliasing. If the runtime behavior shows that the probability of pointer aliasing is low, it is beneficial to speculatively parallelize the loop at runtime.

```
1  for(t=0; t<nt; t++) for(z=0; z<nz; z++)
2    for(y=0; y<ny; y++) for(x=0; x<nx; x++)
3      if(node_number(x,y,z,t)==mynode()){
4        xr=x%squaresize[XUP];
5        yr=y%squaresize[YUP];
6        zr=z%squaresize[ZUP];
7        tr=t%squaresize[TUP];
8        i=xr+squaresize[XUP]
9          *(yr+squaresize[YUP]
10         *(zr+squaresize[ZUP]*tr));
11       lattice[i].x = x;
12       lattice[i].y = y;
13       lattice[i].z = z;
14       lattice[i].t = t;
15       lattice[i].index=x+nx*(y+ny*(z+nz*t));}
```

(a)

```
1  for(i=1; i<n; i++)
2    for(j=iaL[i]; j<iaL[i+1]-1; j++)
3      x[i] = x[i] - aL[j] * x[jaL[j]];
```

(b)

```
1  void VectorAdd(int n,
2           float *c, float *a, float *b)
3    for(int i=0; i<n; i++)
4      *(c + i) = *(a + i) + *(b + i);
```

(c)

Fig. 1. Code examples for (a) nonlinear array access, (b) indirect array access, and (c) array access through pointer.

As described in these examples, loops that are not possible to parallelize at compile time must be reinvestigated at runtime. For loops that have cross-iteration dependencies with low probabilities, speculatively parallelizing loops on the GPU will yield a great performance speedup.

3. PARAGON OVERVIEW

The main goal of Paragon's execution system is to automatically extract fine-grain data parallelism from its sequential input code and generate efficient C/CUDA code to run on a heterogeneous system consisting of a CPU and GPU. However, applications with irregular or complex data dependencies are hard or even impossible to parallelize at compile time. To overcome this problem, Paragon detects possibly parallel loops and runs them speculatively on the GPU. As with any speculation system, two mechanisms are required: a check-pointing state to enable execution rollback and runtime dependence checking to identify miss-speculations.

Paragon utilizes a check-pointing mechanism that is tailored for GPU-enabled systems. Traditionally, at each checkpoint, before starting speculative kernel execution, the speculative execution system takes a snapshot of the architectural state. Storing copies of a few registers at transaction threads in a CPU core is relatively cheap. For GPUs, however, with thousands of threads running, naively check-pointing large register files would incur significant overhead [Fung et al. 2011]. Therefore, it is not practical to use traditional CPU check-pointing mechanisms on the GPU.

Since GPUs and CPUs have separate memory systems, there is no need for special check-pointing before launching a speculative kernel on the GPU. Paragon always keeps

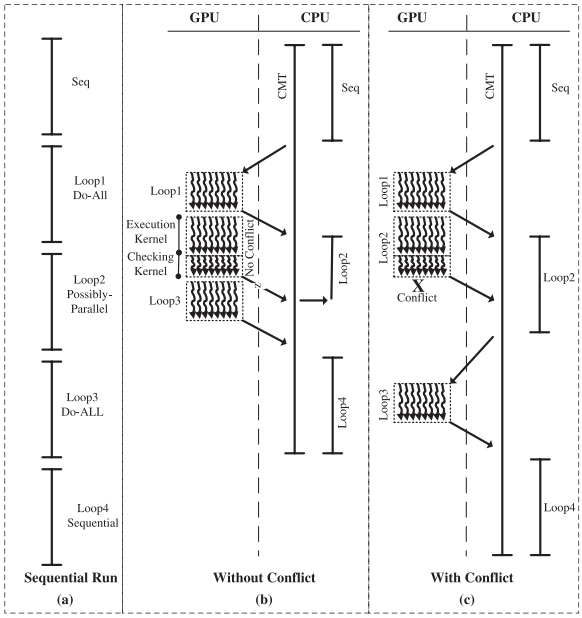

Fig. 2. An example of running a program with Paragon: (a) sequential run, (b) execution without any conflict, and (c) execution with conflict.

one version of the correct data in the CPU's memory, and in case of conflict, it uses the CPU's data to recover. To reduce the overhead of recovery, Paragon uses cooperative execution. Instead of waiting for a speculative kernel to finish and run the recovery process if it is needed, Paragon runs the safe sequential version of the kernel on the CPU in parallel to the GPU version. If there was a conflict in the speculative execution on the GPU, Paragon ignores the GPU's results and waits for the safe execution to finish and uses its result to run the next kernel. On the other hand, if there was not any conflict, Paragon terminates the CPU execution after the GPU kernel is finished successfully. Cooperative execution is key to achieving good performance in Paragon.

The second speculation mechanism is runtime dependence checking to identify miss-speculations. Bulk tracking of memory dependences using signatures along with dedicated structures works well for CPUs with limited numbers of threads. However, for tracking memory accesses of thousands of threads, large signatures per thread are needed. Maintaining and accessing these large signatures dramatically degrades the performance on the GPU. Also, many of these traditional conflict detection approaches need a fast communication mechanism between the cores, which is not available in GPUs. Therefore, Paragon uses a distributed conflict detection mechanism that can check memory accesses of many threads in parallel. This conflict detection mechanism is done in two phases. In the first phase, Paragon updates the write log and read log for each memory access. Then, Paragon checks the write log and read log to detect any conflicts. Both of these phases are specifically designed to utilize the data-parallel power of the GPU to reduce the overhead of conflict detection.

Figure 2 shows an example of Paragon's execution for a program with five different code segments. Like most programs, this program starts with a sequential code. There are four loops with different characteristics in this example. *Loop1* and *Loop3* are parallel. *Loop2* is a possibly parallel loop that has complex or data-dependent cross-iteration dependency, so the compiler is unable to guarantee the safe parallel execution

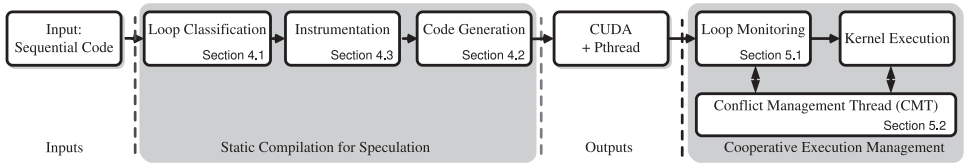

Fig. 3. Compilation flow in Paragon.

of this loop. Finally, *Loop4* has cross-iteration dependencies and is statically classified as a sequential loop. Paragon launches a Conflict Management Thread (CMT) on the CPU. The CMT is responsible for orchestrating GPU-CPU transfers, running kernels on the GPU or CPU, and managing the cooperative execution between the CPU and GPU for speculative kernels. In order to run a kernel on the CPU, the CMT launches another thread, called the *working thread,* on the CPU.

In this example, Paragon starts the execution by running the sequential part on the CPU. After running the sequential code, Paragon transfers the data needed for the execution of *Loop1* to the GPU and starts the parallel version of *Loop1*. Since *Loop2* is possibly parallel, it should be speculatively executed on the GPU. In order to keep the correct data at this checkpoint, Paragon transfers data to the CPU. For *re-entrant* loops that do not update their input arrays, using asynchronous concurrent execution, Paragon launches the CUDA kernel for *Loop2* at the same time. If *Loop2* reads and writes to the same array (i.e., nonre-entrant), Paragon should wait for the data to be completely transferred to the CPU and then launch the GPU kernel. The CPU executes the safe and sequential version of *Loop2* after it receives the data needed for execution of *Loop2* from the GPU. Paragon checks for conflicts in the speculative execution of possibly parallel loops such as *Loop2*. The conflict detection process is done in parallel on the GPU with two kernels: the *execution kernel* and *checking kernel*. The execution kernel executes the loop and also marks addresses accessed by this loop. The checking kernel investigates all these addresses in parallel to detect conflicts and will set a conflict flag if it detects any dependency violation. After *Loop2* is finished, the GPU transfers the conflict flag to the CPU. Based on the conflict flag, there are two possibilities: first, if there was no conflict (Figure 2(b)), the CMT stops the working thread that is executing *Loop2* on the CPU and uses the GPU data to start *Loop3*. The second case is when a conflict is found in parallel execution of *Loop2* as shown in Figure 2(c). In this case, Paragon waits for the CPU execution to finish, then transfers data needed for the *Loop3* to the GPU. Since *Loop3* is a do-all loop, this loop will be executed only on the GPU without speculation. In order to run the sequential *Loop4*, Paragon copies the output of *Loop3* to the CPU.

Figure 3 shows the overall flow of Paragon's compilation and runtime system.

4. COMPILING FOR DATA-PARALLEL SPECULATION

One of the main challenges in Paragon is how to perform lightweight speculation and conflict detection on a massively data-parallel engine similar to a GPU. Traditional approaches for performing speculation on a multicore system fall short in this context due to the vast number of active threads, complex memory architecture, and communication and synchronization overheads in GPUs. Therefore, Paragon is equipped with lightweight data-parallel speculation and distributed conflict detection engines to address these issues.

Paragon focuses on loops in sequential C/C++ applications. As shown in Figure 3, Paragon first performs loop classification to determine which code segments are safe to parallelize. Based on this information, loop classification categorizes each loop into

one of the following three categories: parallel (do-all), sequential, and *possibly parallel*. Parallel loops do not have any cross-iteration dependency and can be run in parallel on the GPU. Sequential parts, which will be run on the CPU, are parts that do not have enough parallelism to run on the GPU or have system function calls. Loops that static analysis cannot determine if they are parallel or sequential will be in the last group called possibly parallel loops.

Loop classification passes all this information to the code generation and instrumentation units. Since the sequential loops will be run on the CPU, Paragon generates only C code for such loops. For parallel loops, CUDA kernels will be generated. Code generation generates the CPU and GPU code with instrumentation for possibly parallel loops. The purpose of the instrumentation is to detect any possible conflict in the execution of unsafe kernels. This distributed conflict detection mechanism has two kernels: the execution kernel and the checking kernel. These two kernels and instrumentations that need to be added will be discussed in Section 4.3. Before that, in the next two sections, Paragon's loop classification and code generation are explained.

4.1. Loop Classification

Loop classification categorizes each loop into one of the following three categories: parallel (do-all), sequential, and possibly parallel. Paragon is using static analyses and transformations such as scalar and array privatization, symbolic data dependence testing, reduction recognition, and induction variable substitution to detect parallel loops [Wolfe 1995].

Besides detection of parallel loops using static analyses, Paragon also searches for sequential loops with indirect, nonlinear, or pointer accesses that may be parallel and marks them as possibly parallel loops. The rest of the loops will be marked as sequential loops. Loop classification sends this information to the next stages, which are kernel generation and instrumentation for conflict detection.

4.2. Kernel Generation

Distributing the workload evenly among thousands of threads is the main key to gaining good performance on a GPU. How to assign loop iterations to threads running on the GPU is a significant challenge for the compiler. This section illustrates how Paragon distributes iterations of the loop among GPU threads.

For single do-all loops, Paragon assigns the loop's iterations to the GPU's threads based on the trip count. If the trip count is fixed and it is smaller than the maximum number of possible threads, Paragon assigns one iteration per thread. Since our experiments show that the best number of threads per block is constant (for our GPUs, it is equal to 256), the number of Threads per Block (TpB) is always equal to 256. Therefore, the number of blocks will be equal to the trip count divided by 256. This number can be easily changed based on the GPU for which Paragon is compiling. If the trip count is more than the maximum possible number of threads, Paragon assigns more than one iteration per thread. The number of Iterations per Thread (IpT) is always a power of two to make it easier to handle on the GPU. In this case, number of blocks (B) will be:

$$B = \frac{Trip\_Count}{TpB * IpT}.$$

On the other hand, if the trip count is not known during the compile time, the compiler cannot assign a specific number of iterations to each thread. In this case, Paragon sets the number of blocks to a predefined value, but this number will be tuned based on the previous runs of this kernel. As shown in Figure 4(b), each thread will run iterations until no iterations are left. We could use this method for loops with fixed

```
1 #pragma unroll
2 for (i=0; i<iterationsPerThread ; i++)
3  perform iteration #(i * blockDim + threadId)
```

(a)

```
1 for (i=threadId ; i<tripCount ; i+=blockDim)
2  perform iteration #(i)
```

(b)

Fig. 4. Generated CUDA code for parallel loops with (a) fixed trip count, and (b) variable trip count.

trip counts, but our experiments show that assigning the exact iterations per thread increases the performance for these loops. If the number of threads launched is less than the number of iterations, some threads will be idle during the kernel execution, and that may degrade the performance. Another advantage is that for loops similar to the loop in Figure 4(a), which has a fixed trip count, the compiler can unroll the loop efficiently.

Nested do-all loops will be easy to compile if Paragon can merge those loops and generate one do-all loop. However, it is not always possible. For imperfectly nested loops, in which all assignment statements are not contained in the innermost loop, it is hard to merge nested loops. In these cases, Paragon merges nested loops as far as it is possible. Finally, two loops will be mapped to the GPU. The outer loop will be mapped to the blocks, and the inner loop will be mapped to threads of blocks. Therefore, the number of blocks will be equal to the trip count of the outer loop and the number of threads per block is still equal to 256.

Reduction loop. A reduction operation generally takes a large array as its input, performs computations on it, and generates a single element as its output. This operation is usually parallelized on GPUs using a tree-based approach, such that each level in the computation tree gets its input from the previous level and produces the input for the next level. In a uniform reduction, each tree level reduces the number of elements by a fixed factor and the last level outputs one element as the final result. The only condition for using this method is that the reduction operation needs to be associative and commutative.

Paragon automatically detects reduction operations in its input using reduction variable analysis [Wolfe 1995]. After this detection phase, the compiler replaces the reduction loop with a highly optimized kernel in its output CUDA code. Paragon uses the optimized CUDA version of the reduction kernel as described in different studies such as the work proposed by Roger et al. [2007].

If there are multiple do-all loops and the innermost loop is a reduction loop, Paragon compiles them based on the trip count of the outer loops. If the trip counts of the outer loops are low, Paragon maps the outer loops to the blocks and each block executes the reduction loop. On the other hand, if the outer loops have a high number of iterations, Paragon may assign each reduction process to one thread. Therefore, iterations of the outer loops will be distributed among threads, and each thread executes one instance of the innermost loop.

After generating CUDA codes for parallel and possibly parallel loops, Paragon inserts copying instructions between the kernels. All live-in and live-out variables for all kernels are determined by Paragon at compile time. After each kernel, Paragon inserts copy instructions based on the previous and next kernel's types. If both consecutive kernels are parallel or sequential, there is no need to transfer data. If one of them is

parallel and the other one is sequential, transferring data is needed. In cases where at least one of the kernels is possibly parallel, Paragon adds copy instructions in both directions: from the CPU to the GPU and from the GPU to the CPU. Cooperative execution management will decide how to move the data at runtime based on the place of correct data.

4.3. Instrumenting for Conflict Detection

One of the main challenges for speculative execution on the GPU is designing a conflict detection mechanism that works effectively for thousands of threads. Traditional techniques used for multicore CPUs are not well suited for GPUs because of the nontraditional memory hierarchy, different synchronization tradeoffs on GPUs, and vast number of active threads available at runtime.

To deal with these constraints, we designed a distributed conflict detection mechanism in our system. Paragon detects the dependencies between different iterations of possibly parallel loops with two kernels: the execution kernel and the checking kernel. The first kernel executes the computations and also tags load and store addresses, and the checking kernel inspects these addresses to find a conflict. In this case, a conflict means writing to the same address by multiple threads or writing to an address by one thread and reading the same address by other threads.

Execution kernel. For indirect and nonlinear array accesses, Paragon detects the arrays that can cause conflicts, and for those arrays, it allocates write-log and read-log arrays. Traditionally, Bloom filters have been used to track the dependencies between threads with very low overhead. However, using a Bloom filter for keeping track of thousands of threads at the same time requires large signatures [Bloom 1970]. Furthermore, accessing these signatures on the GPU requires uncoalesced accesses, which leads to the performance degradation on the GPU. Therefore, instead of using a Bloom filter, Paragon stores all memory accesses in read-log and write-log arrays separately. During the execution, each store to a conflict candidate array will be added to a write log and each load from that array will be added to a read log.

Since the order of execution of threads on the GPU is not known a priori, any two threads that write to the same address can potentially cause a conflict. This conflict may result in a wrong output. Therefore, if the number of writes to one address is more than one, there is considered a write-write dependency violation and that loop is not parallelizable. To detect write-write dependencies, Paragon utilized two approaches:

—**Atomic:** In this approach, Paragon uses CUDA atomic increment instructions to increment the number of writes for each store in a kernel. This method used GPU-specific atomic instruction. Based on the values stored in the read and write logs, the checking kernel can detect dependency violations. Figure 5 shows an example of using the atomic approach. Figure 5(a) shows the original code and Figure 5(b) shows the execution kernel using atomic operation. Since each iteration modified x[i] and also reads x[jal[j]], these accesses to array x can cause conflicts. Therefore, Paragon instruments all accesses to array x. In order to prevent false-positive conflict detection, Paragon just sets the write log for x[i] not the read log. As shown in this example, if Paragon statically detects that one thread accesses the same element several times, it just keeps track of one of the accesses.

—**Reduction:** In this approach, each thread sets the addresses of writes in the write-log array and also each thread counts the number of addresses that it modifies and stores this number in the total-writes array (*tw*) as shown in Figure 6(a). The checking kernel then compares these numbers to detect any possible dependency violations. This approach is a variation of LRPD [Rauchwerger and Padua 1999], which is employed in multicore CPUs.

```
1  for(i=1; i<n; i++)
2    for(j=iaL[i]; j<iaL[i+1]-1; j++)
3      x[i] = x[i] - aL[j] * x[jaL[j]];
```

(a)

```
1  Execution_Kernel()
2    initialize sharedSum to zero
3    for (i=blockIdx.x; i<n ; i+=gridDim.x){
4     sum = 0;
5     for (j=jaL[i]+threadIdx.x;j<iaL[i+1]-1;
6                   j+=blockDim.x){
7       sharedSum[j] += aL[j] * x[jaL[j]];
8       rd_log[jaL[j]] = 1;
9     }
10    sum = compute_sum(sharedSum);
11    if (threadIdx.x == 0){
12      x[i] -= sum;
13      AtomicInc(wr_log[i]);}
14   }
```

(b)

```
1  Checking_Kernel()
2    tid = blockIdx.x * blockDim.x + threadIdx.x;
3    wr = wr_log[tid];
4    rd = rd_log[tid];
5    conflict = wr >> 1 | (rd & wr);
6    if (conflict)
7      conflictFlag = 1;
```

(c)

Fig. 5. Generated CUDA code for example code: (a) with atomic approach, (b) the execution kernel code with instrumentation, and (c) the checking kernel.

In addition to the output dependency, writes to and reads from the same address by two different threads may violate the dependency constraints, and the GPU's result may not be valid anymore. In this case, one read is sufficient to cause a conflict and invalidate the results. Therefore, for decreasing the overhead of maintaining a read-log array, Paragon does not increment read-log elements atomically. Instead, it just sets the corresponding bit in the read log for each read without using any atomic instruction.

The execution kernel is different for loops with pointer accesses, because these loops access memory through pointers and, statically, it is not clear which array they access. In order to keep track of the arrays that each kernel accesses, Paragon stores the start address and size of arrays that are statically allocated on the CPU in a global table. A similar table is also loaded into the GPU's memory. In order to find the arrays corresponding to each of the input pointers, Paragon compares the address of each of the kernel's input pointers with the start addresses and sizes of all the allocated arrays before launching the kernel. Afterward, Paragon transfers these arrays to the GPU memory before launching the kernel. If the array that the pointer accesses is not found in the address table, the pointer is accessing dynamically allocated arrays and Paragon will run these kinds of loops sequentially on the CPU. Moreover, Paragon assumes that each pointer accesses only one array during the kernel execution. Therefore, if a pointer accesses more than one array, Paragon will detect that and raise the conflict flag.

Also, Paragon translates the pointers from the CPU's memory address space to the GPU's memory address space. In order to do this translation, Paragon subtracts the start address of the CPU array from the pointer address and adds it to the start address

```
1  Execution_Kernel()
2    initialize sharedSum to zero
3    write_count = 0;
4    for (i=blockIdx.x; i<n ; i+=gridDim.x){
5     sum = 0;
6     for (j=jaL[i]+threadIdx.x;j<iaL[i+1]-1;
7                   j+=blockDim.x){
8       sharedSum[j] += aL[j] * x[jaL[j]];
9       rd_log[jaL[j]] = 1;
10    }
11    sum = compute_sum(sharedSum);
12    if (threadIdx.x == 0){
13      x[i] -= sum;
14      wr_log[i] = 1;
15      write_count ++;}
16   }
17   tw[thread_id] = write_count;
```

(a)

```
1  Distinct_Writes = compute_sum(wr_log);
2  Total_Writes = compute_sum(tw);
3
4  Checking_Kernel()
5    tid = blockIdx.x * blockDim.x + threadIdx.x;
6    if (tid == 0)
7      if (Total_Writes != Distinct_Writes)
8        conflictFlag = 1;
9    wr = wr_log[tid];
10   rd = rd_log[tid];
11   conflict = (rd & wr);
12   if (conflict)
13     conflictFlag = 1;
```

(b)

Fig. 6. Generated CUDA code for example code in Figure 1(b) with reduction approach: (a) the execution kernel code with instrumentation and (b) the checking kernel.

of the corresponding GPU array. Similarly, Paragon translates these pointers from the GPU to the CPU after the execution of the kernel.

For loops with pointer accesses, it is hard to detect which arrays may cause conflicts. Therefore, Paragon allocates one write-log array and one read-log array whose size is equal to the sum of the sizes of arrays that this loop accesses. This size can be calculated based on the address table. Each array has its own range in the write-log and read-log arrays. At the beginning of the kernel, Paragon again detects which array each pointer accesses and determines its corresponding address in the write-log and read-log arrays as shown in Figure 7(a). Each pointer's address is compared to the beginning and finishing addresses of all arrays, which are stored in *GPU_Table*, to find the corresponding array. By doing this, Paragon is able to detect conflicts when two or more pointers access the same array. Since this process is done at the beginning of the kernel and outside of the main loop, its overhead is small for the kernels that have high trip count loops.

Checking kernel. After completion of the execution kernel, the checking kernel will be launched. This kernel investigates the read log and write log to find conflicts. To find write-read conflicts, the easiest implementation of the checking kernel is to check all addresses as shown in Figure 5(c).

```
1 int Find_Array(Pointer p)
2   for (index = 0: number_of_Arrays)
3      diff =  p - GPU_Table[index].begin;
4      if (diff=>0 & diff<GPU_Table[index].size)
5        return index;
6   return -1;
```

(a)

```
1 bool Range_Check(Pointer Max, Pointer Min, int p_Array)
2  begin = GPU_Table[p_Array].begin;
3  size =  GPU_Table[p_Array].size;
4  if (Min >= begin)
5    & Max < size + begin)
6    return True;
7  else
8    return False;
```

(b)

Fig. 7. CUDA functions that Paragon uses to check pointer memory accesses: (a) finding array that each pointer accesses, which is called outside the main loop; (b) for each pointer, Paragon computes the minimum and maximum addresses that are accessed through that pointer. The range-check function checks these maximums and minimums at the end of the execution kernel to see if all accesses were to the corresponding array or not.

For the reduction approach, Paragon calculates the sum of writes performed by all threads and number of distinct writes performed as follows:

$$Total\_Writes = \sum_{i=0}^{Threads} tw[i].$$

$$Distinct\_Writes = \sum_{i=0}^{Addresses} write\text{-}log[i].$$

These two sums are computed using reduction kernels as shown in Figure 6(b). If *Total_Writes* is more than *Distinct_Writes*, it means that two or more threads write to the same address, which is an output conflict.

Since the exact number of writes to each address is known in the atomic approach, there is no need to launch reduction kernels. Line 5 of Figure 5(c) checks the number of writes and reads of the corresponding element. If the number of writes is more than one ($wr \gg 1$) or there is at least one write and one read ($rd \,\&\, wr$), the checking kernel will set the conflict flag.

However, checking all addresses may degrade the performance. Instead, it will be advantageous to just check those addresses that at least one of the execution kernel's threads writes to. In order to check these addresses, the checking kernel should regenerate indices that threads of the execution kernel wrote to them. For each store, Paragon starts from the index of the store instruction and traverses the dataflow graph in the reverse order, to build up a slice of instructions on which the store depends, either directly or indirectly. This process stops when it reaches the input variables or the loop indices. The checking kernel executes these instructions to regenerate the store indices and investigates them to find a conflict.

Paragon uses an optimization for loops in which write-write dependencies are the only possible source of conflicts. In these types of loops, there is no need to launch the checking kernel because the *atomicInc* function returns the old value of the write-log element. For each write that may cause conflict, the execution kernel increments the

corresponding element in the write-log array and it also checks the old value. If the old value is more than zero, it shows that another thread already wrote to the same element. In such a case, this access is marked as a conflict.

Since atomic instructions are slower than nonatomic memory accesses, the execution kernel runs faster for the reduction approach. However, as mentioned before, the reduction approach needs to calculate the sum of two arrays, write log and total writes, before executing the checking kernel. Since these two reduction operations have a significant overhead for large arrays, checking for conflicts in the atomic approach has lower overhead than in the reduction approach.

For loops with pointer accesses, Paragon runs the same checking kernel as Figure 5(c), but it also takes an additional step to make sure no cross-array dependency violation is happening. Paragon assumes that each pointer accesses only one array during the execution of a kernel. Therefore, for kernels with pointers that may access multiple arrays, Paragon raises the conflict flag. In order to detect these pointers, Paragon keeps track of the maximum index and the minimum index that each pointer accesses. By computing maximum and minimum with the max and min intrinsic functions available in CUDA, this range check process is done without any dataflow divergences. At the end of the kernel, all these maximums and minimums will be checked to see if each pointer accesses only one array or not as shown in Figure 7(b). If Paragon detects that a pointer accesses different arrays during the kernel execution, it stops the GPU execution and transfers the execution to the CPU.

Whenever Paragon finds a conflict, it will set the conflict flag. This flag will be sent to the CPU and, based on that, the CMT makes further decisions. These decisions will be discussed in Section 5.

5. COOPERATIVE EXECUTION MANAGEMENT

The cooperative execution management unit in Paragon is a runtime component that is in charge of deciding where a loop should execute, coordinating execution of a possibly parallel loop between the CPU and GPU, and orchestrating data transfers between the host and GPU memories. Paragon tries to increase the efficiency of speculation by utilizing both the GPU and CPU at the same time. This cooperation between the CPU and GPU can reduce the overhead of speculation in case of miss-speculations.

During runtime, the first invocation of possibly parallel loops will be monitored to find any dependency between different iterations. After loop monitoring, possibly parallel loops will be categorized as a parallel or sequential loop based on the number of dependencies found in the monitoring result. Sequential loops will be run on the CPU, and parallel loops will be run speculatively on the GPU. For speculative execution on the GPU, Paragon requires the original code to be augmented with the instructions that drive the runtime dependence analysis.

The main unit of cooperative execution management is the CMT, which uses monitoring information to decide which kernels should be executed on the GPU and which of them should be run on the CPU. The CMT also takes care of data movement between the CPU and GPU especially in miss-speculation cases.

5.1. Loop Monitoring

This section describes how Paragon monitors possibly parallel loops on the CPU to find the dependency between iterations and uses this information to improve the performance of the generated code. Paragon executes the first invocation of possibly parallel loops on the CPU with two threads: *working thread* and *monitoring thread*. The working thread executes the loop sequentially and the monitoring thread monitors the loop in parallel to decrease the overhead of monitoring. The monitoring thread keeps track

of all memory accesses. This one-time monitoring has a negligible overhead because Paragon only monitors possibly parallel loops in parallel with the real execution.

The monitoring thread executes every instruction from the loop except stores and keeps track of the number of conflicts. After monitoring each possibly parallel loop, if there was no conflict (read-after-write, write-after-read, and write-after-write), the monitoring thread marks the kernel as a parallel kernel for the CMT. If there were conflicts, the monitoring thread marks the loop as sequential. After all possibly parallel kernels are categorized based on the monitoring results, Paragon enters the kernel execution phase. In this phase, Paragon keeps track of the number of iterations that each loop has. Based on these numbers, it will tune the number of blocks for the next execution of each kernel on the GPU to get the best performance.

5.2. Conflict Management Thread (CMT)

The Conflict Management Thread (CMT) is a thread running on the CPU and its responsibility is to manage GPU-CPU transfers and run kernels speculatively on the GPU. The CMT decides which kernel should be executed on the CPU or GPU. In case of conflicts, it uses the correct data on the CPU to run the next kernel. If there was a dependency violation, the CMT does not launch the next kernel on the GPU and waits for the working thread on the CPU to finish. Based on the next kernel type, the CMT makes different decisions. If the next kernel should be run on the GPU, the CMT transfers all live-out variables to the GPU and launches the next kernel. If the next kernel is possibly parallel, in addition to the GPU version, one version will also be run on the CPU. The last case is that the next kernel is sequential, so the CMT runs the sequential code on the CPU.

If there was no conflict in the GPU execution, the CMT sets a global variable to inform the working thread on the CPU to stop. To decrease the overhead, the working thread checks that global variable once every several iterations (10 in our experiments). This global variable works as a memory barrier to manage the data transferring between the CPU and GPU. If the next kernel is parallel, the CMT will launch the next kernel. Otherwise, it transfers live-out variables and runs the next kernel on the CPU.

5.3. Execution Scenarios

This section explains the advantages and disadvantages of using cooperative loop execution for different possible scenarios. Figure 8 shows four different possibilities for an example with three loops. In this example, *Loop2* is a possibly parallel loop, and based on the characteristics of the *Loop1* and *Loop3*, different scenarios may take place. In the first scenario shown in Figure 8(a), *Loop1* and *Loop3* are sequential loops and they will be executed on the CPU. Figure 8(b) shows a case when both *Loop1* and *Loop3* are do-all loops and will be run on the GPU. In Figures 8(c) and 8(d), one of these two loops is do-all and the other one is sequential.

For each of these scenarios, there are three cases: the first case is the baseline when Paragon does not run the possibly parallel loop (*Loop2*) speculatively on the GPU. In all baseline cases, *Loop2* will be executed on the CPU. In the next two cases, Paragon runs the loop speculatively on the GPU. If there was no conflict in the GPU execution, Paragon uses the GPU's results and launches the next kernel. In the last case, there is a conflict in the GPU execution. Therefore, Paragon continues the CPU version of *Loop2* and uses its results to execute the last loop.

Figure 8 compares the execution time (τ) of *Loop2,* which includes the transfer times needed for executing this loop. In other words, τ is equal to the time between the termination of the first loop and the start of the last loop. L is the execution time of *Loop2* on the CPU and G is the speedup of the GPU execution of *Loop2* compared to the sequential run of *Loop2*. For the sake of simplicity, it is assumed that transfer time

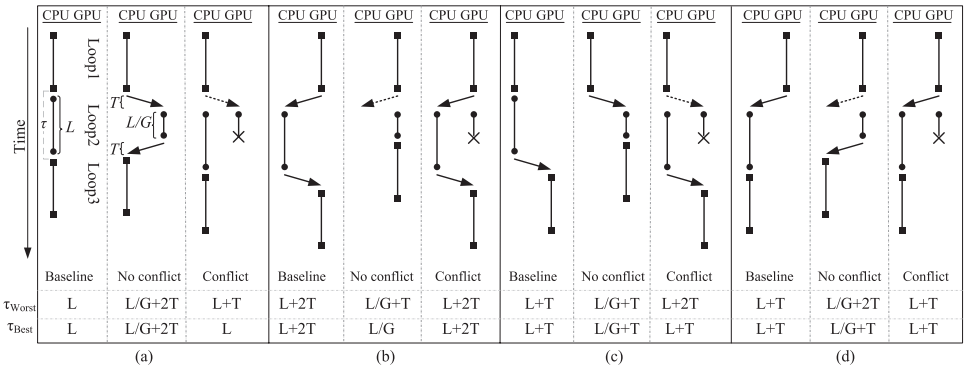

Fig. 8. Different scenarios for Paragon execution. This figure compares the execution time (τ) of *Loop2* for different scenarios. τ is equal to the time between termination of the first loop and the start of the last loop. L is the execution time of the *Loop2* on the CPU and G is the speedup of the GPU execution of the *Loop2* with instrumentation compared to the sequential CPU execution. T is the transfer time between the CPU and the GPU.

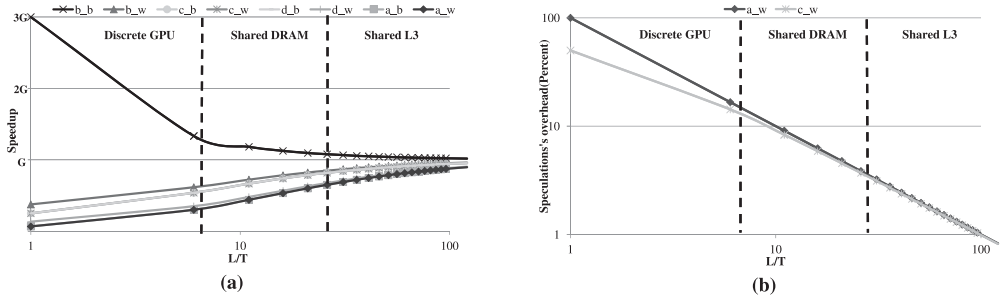

Fig. 9. This figure shows performance and speculative overhead for different execution scenarios in Figure 8. Part (a) illustrates speedup of different scenarios compared to the baseline when there is no conflict. Scenario b in the best case (b_b) has the highest speedup and scenario a in the worst case (a_w) has the lowest speedup. All the legends are sorted based on the speedup on top of the figure. Part (b) illustrates the overhead of these scenarios compared to the baseline in case of miss-speculation.

from the GPU to the CPU is equal to the transfer time from the CPU to the GPU, which is equal to T.

If *Loop2* does not modify the inputs of the loop, there is no need to wait for the transfer operation to be over, and Paragon can perform the transfer and launch the kernel at the same time. For example, in Figure 8(b), with no conflicts, if *Loop2* is re-entrant, the GPU version can start right after *Loop1*. However, if *Loop2* is not re-entrant, Paragon transfers the data to the CPU before starting *Loop2* on the GPU to make sure that there is a correct version of the data in the CPU's memory. Dashed arrows in Figure 8 represent these kinds of transfers, which, based on the characteristics of the possibly parallel loop, may or may not affect the execution time.

Figure 9(a) shows the speedup that Paragon can gain with speculation for different L/Ts. This speedup is equal to $\tau_{baseline}/\tau_{noconflict}$. In fused architectures where the CPU and the GPU are integrated on the same die and share DRAM, as in AMD Fusion, or L3 cache, as in Intel Sandy Bridge, transfer time is low compared to the discrete GPUs.[1] As can be seen in the figure, speedup for these systems will be close to the

[1]A typical discrete GPU has a separate memory from system memory and data transfer is done through PCIExpress.

Table I. Application Specifications

	Input Size	Output Size	Number of loops
FDTD	4096 × 4096 matrix	4096 × 4096 matrix	6
Seidel	4096 × 4096 matrix	4096 × 4096 matrix	2
Jacobi1d	16M array	16M array	1
Jacobi2d	4096 × 4096 matrix	4096 × 4096 matrix	2
Gemm	two 4096 × 4096 matrix	4096 × 4096 matrix	3
Tmv	4096 × 4096 matrix + 4096 array	4096 array	2
Saxpy	two 32M array	32M array	1
House	two 32M array	32M array	2
Ipvec	32M array	32M array	1
Ger	two 64K array + sparse 64k × 64k matrix	sparse 64k × 64k matrix	2
Gemver	two 64K array + sparse 64k × 64k matrix	64k array	6
FWD	64K array + sparse 64k × 64k matrix	64k array	2
SOR	64K array + sparse 64k × 64k matrix	64k array	2

GPU's gain (G) in all scenarios. The interesting point in this figure is that speedup is increasing by decreasing the transfer time, except in scenario (b) with the re-entrant loop (the best case). The reason for that is *Loop1* and *Loop3* are both executed on the GPU. In the baseline case, Paragon should transfer the input data of *Loop2* to the CPU, execute that loop, and transfer the data back to the GPU. For speculative execution, there is no need to transfer the data. Therefore, reducing the transfer time will reduce the advantage of speculation over baseline for this scenario.

Figure 9(b) shows the overhead of miss-speculation for scenarios 8a and 8c in the worst case (nonre-entrant loop) for different L/Ts. All other scenarios do not have any performance overhead in case a conflict happens. With decreasing transfer cost, this overhead decreases rapidly as shown in this figure.

6. EXPERIMENTS

Paragon compilation phases are implemented in the backend of the Cetus compiler [Lee et al. 2003]. We modified the C-code generator in Cetus to generate CUDA code. Paragon's output codes are compiled for execution on the GPU using NVIDIA nvcc 4.0. GCC 4.4.6 is used to generate the x86 binary for execution on the host processor. The target system has an Intel i7 CPU and an NVIDIA GTX 560 GPU with 2GB GDDR5 global memory.

In order to evaluate Paragon, we compiled benchmarks with pointer and indirect memory accesses and compared their performance with hand-optimized unsafe parallelized C code.[2] We implemented unsafe parallel versions of these benchmarks for the CPU with two and four threads and for the GPU too. Although there are many works on speculation for CPUs like CorD [Tian et al. 2008], their performance cannot be better than unsafe parallel versions. For example, CorD has 7% overhead. That's why we use unsafe code as an upper bound in our performance measurements for comparison purposes. A summary of the benchmark characteristics is shown in Table I. Also, we present a case study of accelerating a real-world application, Rayleigh quotient iteration, which will be discussed in Section 6.4.

[2]Unsafe means sequential code that is optimistically parallelized and does not perform any dynamic dependence checking or synchronization/locking to ensure correct results. Therefore, its final results might be wrong due to memory dependencies between threads.

Benchmarks with pointer memory accesses. We reimplemented six benchmarks from the Polybench benchmark suite [Polybench 2011] in C with pointers to show Paragon's performance for loops with pointers.

FDTD, the finite difference time domain method, is a powerful computational technique for modeling electromagnetic space. This benchmark has three pairs of different stencil loops and all these loops are highly memory intensive. The *Seidel* benchmark uses the Gauss-Seidel method, which is an iterative method used to solve a linear system of equations. Seidel is a stencil benchmark with more computation than FDTD. *Jacobi* is another stencil method to solve linear systems; we used one-dimensional and two-dimensional versions of this benchmark.

Gemm is a general matrix multiplication benchmark that has three nested loops. The innermost loop is a reduction loop and two outer loops are parallel. As mentioned before, Paragon decides which loops should be parallelized based on the number of iterations. Since both outer loops have high trip counts, Paragon parallelizes these loops and executes reduction sequentially inside each thread. It should be noted that this code is automatically generated for matrix multiply with pointers, so most compilers cannot detect that these loops are parallel. For the CPU version, we parallelized the outermost loop.

Tmv is a transposed matrix vector multiplication benchmark that has two nested loops. The outer one is a do-all loop and the inner one is a reduction loop. The outer loop will be mapped to thread blocks and each thread block performs the reduction in parallel.

Benchmarks with indirect memory accesses. Seven benchmarks from the sparse matrix library are used to show Paragon's performance for loops with indirect array accesses. We selected them because they have loops that cannot be analyzed by traditional compilers. For each sparse matrix benchmark, we generated matrices randomly with 1% nonzero elements.

Saxpy adds a sparse array with a dense array and writes the result in the dense array. The householder reflection benchmark, *House*, computes the reflection of a plane or hyperplane containing the origin. This method is widely used in linear algebra to compute QR decompositions. This benchmark consists of two parts. The first part is a reduction loop that cannot cause conflict, and this loop will be compiled to CUDA without any instrumentation. The second part has a loop that is similar to *Saxpy* and it may have cross-iteration dependencies.

Ipvec is a dense matrix benchmark that shuffles all elements of the input array based on another array and puts the results in the output array. Sparse BLAS functions *Ger* and *Gemver* also have loops that can cause conflicts. Dependencies between different iterations of these loops cannot be analyzed statically, so we need to use Paragon to run these loops on the GPU speculatively.

Forward elimination with level scheduling, *FWD*, is another method used in solving linear systems. FWD's code is shown in Figure 1 and it has both reads and writes to the conflicted array. The next benchmark is *SOR*, a multicolor SOR sweep in the EllPack format, and its code is similar to FWD. This benchmark has two loops: the outer loop is do-across and the inner loop is parallel, but traditional static compilers cannot easily detect that.

6.1. Performance

Figures 10(b) and 10(a) compare the performance of the benchmarks with pointer and indirect memory accesses to the unsafe parallel execution. The Paragon-Reduction version is the performance of the Paragon's generated code with instrumentation using reduction to check the dependencies. Paragon-Atomic uses the CUDA atomic instruction

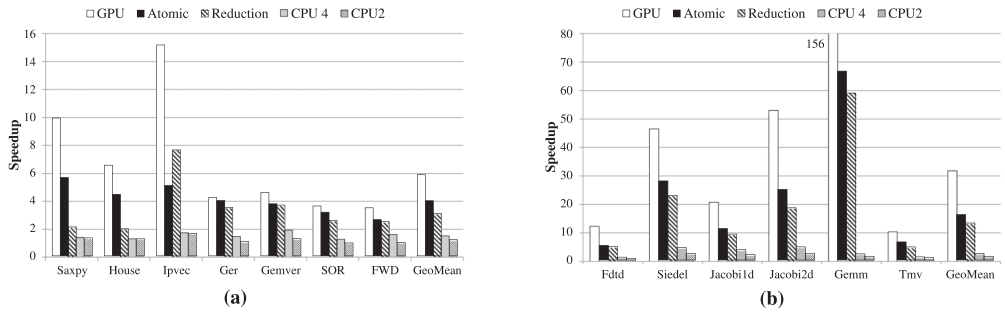

Fig. 10. This figure shows performance of Paragon approaches compared to unsafe parallelized versions. Baseline is running the code sequentially on the CPU. Part (a) illustrates performance comparison of Paragon with unsafe parallel versions on the GPU and CPU with four and two threads for loops with pointers. Part (b) shows performance for loops with indirect accesses.

to find the conflicts on the fly. CPU 4 and 2 are unsafe parallel CPU versions without any checks for conflicts. GPU is an unsafe parallel version of applications without any instrumentations. All these different versions are compared with the sequential runs on the CPU without any threading.

Since memory accesses in benchmarks with indirect accesses are irregular, the GPU's performance is lower for these benchmarks than regular access benchmarks. In these loops, unlike the pointer loops, Paragon marks arrays that can cause conflicts. Since Paragon only checks memory accesses for these arrays, the overhead of conflict checking is lower compared to the pointer loops.

As shown in Figures 10(b) and 10(a), for all benchmarks except *Ipvec*, Paragon-Atomic performs better than Paragon-Reduction. Since atomic instructions are slower than nonatomic memory accesses, maintaining write history of different iterations in Paragon-Reduction has less overhead than Paragon-Atomic. However, in order to find conflicts, Paragon-Reduction needs to calculate the sum of two arrays: write log and total writes. Since these two reduction operations have a large overhead for large array sizes, the performance of the Paragon-Atomic approach is better than Paragon-Reduction. The performance gap between these two approaches is higher for benchmarks with higher checking kernel overhead. Paragon-Reduction performs better than Paragon-Atomic for *Ipvec* because atomic memory accesses perform poorly for the many uncoalesced memory accesses found in *Ipvec*.

For benchmarks with pointer accesses, Paragon-Atomic is 6.8x faster than CPU execution with four threads. For these benchmarks, Paragon is 12x faster than two-thread execution. Also, Paragon-Atomic is 1.3x faster than the Paragon-Reduction approach on average. As can be seen in Figure 10(a), the performance of the Paragon-Atomic is 2.5x better than the unsafe parallel version of the code running on the CPU with four threads for benchmarks with indirect accesses. Paragon-Atomic is 3.4x faster than two-thread execution. Also, Paragon-Atomic is 1.3x faster than the Paragon-Reduction approach on average. It should be noted that in the CPU version, we assumed that there is no conflict between different iterations and, therefore, our results are pessimistic. Figure 10(a) shows that running safely on the GPU is better than running unsafely on the CPU for these data-parallel loops.

On average, for benchmarks with pointer accesses and indirect array accesses, the unsafe parallel GPU versions are respectively 1.9x and 1.5x faster than Paragon-Atomic's performance. The reason is that in loops with pointers, all arrays can cause conflicts, and Paragon's approach can lead to 2x more memory accesses. These extra memory accesses can degrade the performance of memory-intensive loops. The unsafe

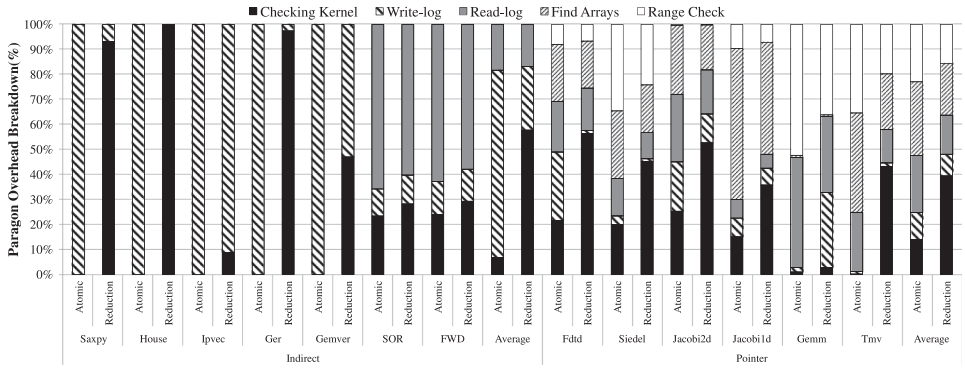

Fig. 11. Breakdown of Paragon's overhead compared to unsafe parallel version on the GPU for loops with pointers.

GPU code's performance is not realistically achievable, and we report this number to show the potential of our system if further optimizations and smarter runtime systems are deployed in Paragon.

6.2. Overhead Breakdown

Figure 11 shows the overhead of Paragon execution compared to the unsafe GPU execution without any instrumentation. This figure also breaks down the overhead into five groups: write-log maintenance, read-log maintenance, checking kernel execution, detecting which arrays each pointer accesses, and range check of indices that each pointer accesses. Note that only benchmarks with pointer memory accesses have find-arrays or range-check overhead.

Saxpy, *House*, *Ipvec*, *Ger,* and *Gemver* only write to the conflict candidate arrays. Since the *atomicInc* function used in the Paragon-Atomic approach returns the old value, there is no need to launch the checking kernel. For each write in the execution kernel, each thread atomically increments the corresponding element in the write log and it checks the old value. If the old value is more than zero, the execution kernel sets the conflict flag. Therefore, for these benchmarks there is no checking-kernel overhead.

For all benchmarks, the write-log overhead is higher for Paragon-Atomic than Paragon-Reduction. The reason is that Paragon-Atomic uses atomic instructions to update the write log that are not as fast as just writing to the global memory. Also, since Paragon-Reduction needs to count the number of writes with executing two reduction kernels, the overhead of the checking kernel is higher for the Paragon-Atomic approach.

SOR and *FWD* benchmarks read from an array and write to the same array with different addresses. Consequently, both Paragon approaches need to launch the checking kernel. Therefore, the overhead breakdown is similar for both approaches.

Benchmarks with pointer memory accesses have range-check and find-arrays overhead too. Find-arrays overhead is negligible for benchmarks with high computation such as *Gemm* because finding arrays is done only once for each kernel. Range-check overhead is high for benchmarks with a large number of memory accesses such as *Gemm* and *Tmv* because for each memory access, Paragon needs to compare the accessed address with maximum and minimum addresses that are accessed by that pointer. Since *Gemm* and *Tmv* have more reads than writes, the overhead of maintaining the read log is higher than the write log's maintenance overhead. The effect of finding arrays is smaller on *Jacobi2d* compared to the same value in *Jacobi1d* because the two-dimensional version has more computation and memory accesses and

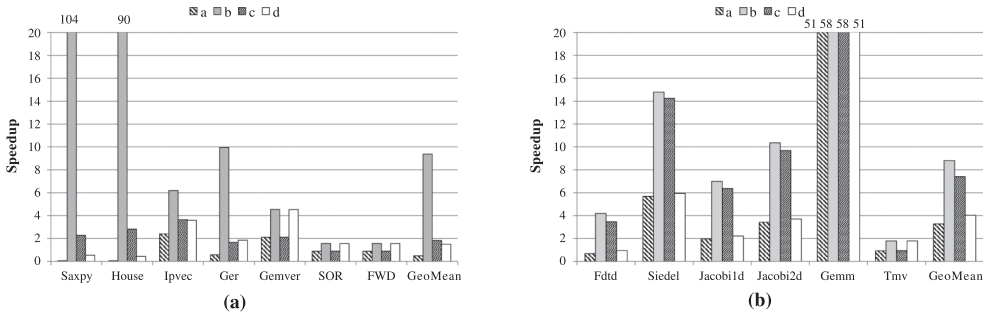

Fig. 12. This figure shows the performance of Paragon for all four different scenarios to the sequential C code. Part (a) illustrates performance for loops with pointers. Part (b) shows performance for loops with indirect accesses.

the find-array process is done completely outside of the loop at the beginning of the kernel.

On average, for the Paragon-Atomic scheme, the overhead introduced by checking kernel is 6% for indirect access and 14% for pointer access benchmarks. As mentioned before, this overhead is higher for the Paragon-Reduction scheme. The checking kernel overhead for Paragon-Reduction is 57% and 39% for benchmarks with indirect and pointer memory accesses, respectively. For benchmarks with indirect memory accesses, maintaining write-log overhead is 74% for Paragon-Atomic, but it is 25% for Paragon-Reduction. Since the only difference between two schemes is how they detect write-write conflicts, the effects of read log, range check, and find arrays are similar for both approaches.

6.3. Execution Scenarios Performance

This section describes the impact of transferring data between the CPU and GPU for different scenarios discussed in Section 5.3. Figures 12(b) and 12(a) compare the performance of the benchmarks with pointer and indirect memory accesses to the sequential C code on the CPU for all four different scenarios. As discussed in Section 5.3, transferring overhead is high for scenario (a) because the previous and next kernel are executed on the CPU. In this case, Paragon transfers the input data to the CPU and transfers the result back. That's why performance improvement for scenario (a) is smaller than the gain reported in Figures 10(a) and 10(b), which do not consider the transferring time.

On the other hand, transferring time helps Paragon to get better speedups for scenario (b). In this scenario, baseline transfers the data from the GPU to the CPU, runs the possibly parallel kernel, and transfers the data back to the GPU. Instead, Paragon executes the possibly parallel kernel on the GPU, and if the loop is re-entrant, there is no need to wait for transferring data. For this scenario, Paragon gets more than 8x speedup for both types of loops on average.

For scenarios (c) and (d), final performance gain is dependant on transferring time for input or output data, and whether the loop is re-entrant or not. For loops with pointer accesses, Paragon cannot decide whether the loop is re-entrant. Therefore, it waits for the transfer. That's the reason that transferring overhead for scenarios (c) and (d) is higher for loops with pointer accesses than loops with indirect accesses.

6.4. Case Study

In this section, we look into the effects of our compiler on the performance of the Rayleigh quotient benchmark. We used this benchmark to demonstrate Paragon's

```
1 rayleigh(A,epsilon,mu,x)
2   x = x / norm(x);
3   y = (A-mu*eye(rows(A))) \ x;
4   lambda = transpose(y)*x;
5   mu = mu + 1 / lambda
6   error = norm(y-lambda*x) / norm(y)
7   while (error > epsilon){
8     x = y / norm(y);
9     y = (A-mu*eye(rows(A))) \ x;
10    lambda = transpose(y)*x;
11    mu = mu + 1 / lambda
12    error = norm(y-lambda*x) / norm(y)
13  }
```

Fig. 13. Rayleigh quotient code.

performance for applications with several loops where a large amount of data has to be shipped back and forth between the GPU and CPU. We also investigate the overhead of Paragon execution in the presence of conflicts in this section. A Rayleigh quotient iteration is an eigenvalue algorithm that extends the idea of the inverse iteration by using the Rayleigh quotient to obtain increasingly accurate eigenvalue estimates.

Rayleigh quotient iteration is an iterative method; that is, it must be repeated until it converges to an answer. Fortunately, very rapid convergence is guaranteed and no more than a few iterations are needed in practice. The Rayleigh quotient iteration algorithm converges cubically for symmetric matrices, given an initial vector that is sufficiently close to an eigenvector of the matrix that is being analyzed.

To solve the linear systems in lines 3 and 9 in Figure 13, we used the biconjugate gradient stabilized method (BiCGSTAB), which is an iterative method used for finding the numeral solution of linear systems such as $Ax = B$ for x where A is a square matrix. The whole BiCGSTAB process can be executed on the GPU.

If matrix A is a sparse matrix, computing *A-mu*eye(rows(A))* will be a possibly parallel code. In this case, a conservative compiler will run this part on the CPU and transfer the result from the CPU to the GPU. However, Paragon speculatively runs this loop on the GPU and removes the transfer overhead. We observed that this loop is executed 5.3x faster on the GPU, and if we consider the transfer time, this speedup will be increased to 7.8x. The effect of this speculation on the whole benchmark is dependant on how accurate linear systems in lines 3 and 9 should be solved.

To show the overhead of Paragon's execution in case of conflict, we added one write-write dependency to every 20 iterations of the speculative kernel. As expected, the performance impact of detecting conflict and using the CPU's data to continue the execution is negligible. Our experiments show that the overhead is less than one percent for this benchmark.

7. RELATED WORK

As many-core architectures have become mainstream, there has been a large body of work, such as SUIF [Wilson et al. 1994] and Polaris [Blume et al. 1996], on static compiler techniques to automatically parallelize applications to utilize thread-level parallelism. These compilers automatically detect loops that can be parallelized using static analyses and transform the loops for parallel execution. However, it is hard to statically decompose the application to take advantage of the growing number of processor cores [Kulkarni et al. 2007, 2009]. One of the most challenging issues in automatic parallelization is to discover loop-carried dependencies. Although various research projects on loop dependence analysis [Psarris et al. 1993] and pointer analysis [Nystrom et al. 2004] have tried to disambiguate dependencies

between iterations, parallelism in most real applications cannot be uncovered at compile time due to irregular access patterns, complex use of pointers, and input-dependent variables.

For those applications that are hard to parallelize at compile time, Thread-Level Speculation (TLS) is used to resolve loop-carried dependencies at *runtime*. In order to implement TLS, several extra compiler and runtime steps such as buffering memory access addresses for each thread, checking violations, and using recovery procedures in case of conflicts between threads are necessary. Software-only approaches [Steffan et al. 2005; Bocchino et al. 2008; Volos et al. 2009; Couceiro et al. 2009; Kim et al. 2012; Kotselidis et al. 2008; Harris et al. 2006; Mehrara et al. 2009; Oancea and Mycroft 2008; Tian et al. 2008] implement all these steps in software. However, most existing proposals for software-only speculative runtimes target tens of cores at most [Mehrara et al. 2009; Oancea and Mycroft 2008; Tian et al. 2008]. Kim et. al. [2012] targets 100 cores, but even their method is not applicable to the GPU because they validate the correctness of all speculative memory accesses on a core in parallel to the loop execution on other cores. However, this parallel check is not possible on the GPU due to communication and synchronization overheads on the GPU.

There are previous works that have focused on generating CUDA code from sequential input [Han and Abdelrahman 2010; Baskaran et al. 2010; Wolfe 2010; Leung et al. 2009; Tarditi et al. 2006]. HiCUDA [Han and Abdelrahman 2010] is a high-level directive-based compiler framework for CUDA programming where programmers need to insert directives into sequential C code to define the boundaries of kernel functions. The work proposed by Baskaran et al. [2010] is an automatic code transformation system that generates CUDA code from input sequential C code without annotations for affine programs. In the system developed by Wolfe [2010], by using C pragma preprocessor directives, programmers help the compiler to generate efficient CUDA code. Tarditi et al. [2006] proposed accelerator, in which programmers use C# and a library to write their programs and let the compiler generate efficient GPU code. The work by Leung et al. [2009] proposes an extension to a Java JIT compiler that executes program on the GPU. Delite [Chafi et al. 2011] is another approach that aims at simplifying the creation of performance-oriented DSLs and compiling them for heterogeneous systems, including systems with GPUs. Our approach is orthogonal to these systems and can be integrated in such compilation frameworks to increase the efficiency of these systems by enabling them to run more applications on the GPU. In order to improve the performance of automatic parallelization, Paragon can take advantage of polyhedral models [Bastoul 2004; Pouchet et al. 2008; Baskaran et al. 2008], which can perform more powerful automatic parallelization.

While none of the previous works on automatic compilation for current GPUs considered speculation, there are other works [Menon et al. 2012; Diamos and Yalamanchili 2010; Liu et al. 2011] that studied the possibility of speculative execution on the GPU. Menon et al. [2012] modified the GPU hardware to support voltage speculation. Diamos and Yalamanchili [2010] described speculative execution on multi-GPU systems exploiting multiple GPUs, but they explored the use of traditional techniques to extract parallelism from a sequential loop in which each iteration launches a GPU kernel. This approach leveraged the possibility of speculatively partitioning several kernels on multiple GPUs. Liu et al. [2011] showed the possibility of using GPUs for speculative execution using a GPU-like architecture on FPGAs. They implemented software value prediction techniques to accelerate programs with limited parallelism, and software speculation techniques, which re-executes the whole loop in case of a dependency violation.

Recent works [Cederman et al. 2010; Fung et al. 2011] proposed software and hardware transactional memory systems for graphic engines. In these works, each thread

is a transaction, and if a transaction aborts, it needs to re-execute. This re-execution of several threads among thousands of threads may lead to control divergence on the GPU and will degrade the performance. For Paragon, each kernel is a transaction, and if it aborts, Paragon uses the CPU's results instead of re-executing the kernels. There are many other works that try to improve the performance of GPUs by different approaches such as reducing the overhead of divergence [Brunie et al. 2012; Coutinho et al. 2011; Zhang et al. 2011], coalescing more memory accesses [Zhang et al. 2011], improving inter-block communication [Xiao and chun Feng 2010], and generating different kernels for different input sizes [Samadi et al. 2012].

8. CONCLUSION

GPUs provide an attractive platform for accelerating parallel workloads. Due to their nontraditional execution model, developing applications for GPUs is usually very challenging. As a result, these devices are left underutilized in many commodity systems. Several languages have emerged to solve this challenge, but past research has shown that developing applications in these languages is a difficult task because of the tedious performance optimization cycle or inherent algorithmic characteristics of an application. Also, previous approaches of automatically generating optimized parallel code in CUDA for GPUs using complex compiler infrastructures have failed to utilize GPUs that are present in everyday computing devices.

In this work, we proposed *Paragon*: a static/dynamic compiler platform to speculatively and cooperatively run possibly data-parallel pieces of sequential applications on GPUs and CPUs. Paragon monitors the dependencies for possibly data-parallel loops running speculatively on the GPU and nonspeculatively on the CPU using a lightweight distributed conflict detection designed specifically for GPUs, and transfers the execution to the CPU in case a conflict is detected. Paragon resumes the execution on the GPU after the CPU resolves the dependency. We looked at two classes of implicitly data-parallel applications: applications with indirect and pointer memory accesses. Our experiment show that, for applications with indirect memory accesses, Paragon achieves 2.5x on average and up to 4x speedup compared to unsafe CPU execution with four threads. Also, for applications with pointer memory accesses, Paragon achieves 6.8x on average and up to 30x speedup compared to unsafe CPU execution with four threads.

ACKNOWLEDGMENTS

Much gratitude goes to the anonymous referees who provided excellent feedback on this work. This research was supported by ARM Ltd. and the National Science Foundation under grants CNS-0964478 and CCF-0916689.

REFERENCES

M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, R. Atanas, and P. Sadayappan. 2008. A compiler framework for optimization of affine loop nests for GPGPUs. In *Proc. of the 2008 International Conference on Supercomputing*. 225–234.

M. M. Baskaran, J. Ramanujam, and P. Sadayappan. 2010. Automatic C-to-CUDA code generation for affine programs. In *Proc. of the 19th International Conference on Compiler Construction*. 244–263.

C. Bastoul. 2004. Code generation in the polyhedral model is easier than you think. In *Proc. of the 13th International Conference on Parallel Architectures and Compilation Techniques*. 7–16.

B. H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM* 13, 7, 422–426.

W. Blume, et al. 1996. Parallel programming with Polaris. *IEEE Computer* 29, 12, 78–82.

R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. 2008. Software transactional memory for large scale clusters. In *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 247–258.

N. Brunie, S. Collange, and G. Diamos. 2012. Simultaneous branch and warp interweaving for sustained GPU performance. In *Proc. of the 39th Annual International Symposium on Computer Architecture*. 49–60.

D. Cederman, P. Tsigas, and M. T. Chaudhry. 2010. Towards a software transactional memory for graphics processors. In *Proc. of the 12th Eurographics Symposium on Parallel Graphics and Visualization*. 121–129.

H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. 2011. A domain-specific approach to heterogeneous parallelism. In *Proc. of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 35–46.

M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. 2009. D2STM: Dependable distributed software transactional memory. In *Proc. of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*. 307–313.

B. Coutinho, D. Sampaio, F. Pereira, and W. Meira. 2011. Divergence analysis and optimizations. In *Proc. of the 20th International Conference on Parallel Architectures and Compilation Techniques*. 320–329.

G. Diamos and S. Yalamanchili. 2010. Speculative execution on Multi-GPU systems. In *Proc. of the 2010 IEEE International Symposium on Parallel and Distributed Processing*. 1–12.

W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt. 2011. Hardware transactional memory for GPU architectures. In *Proc. of the 44th Annual International Symposium on Microarchitecture*.

T. Han and T. Abdelrahman. 2010. hiCUDA: High-level GPGPU programming. *IEEE Transactions on Parallel and Distributed Systems* 22, 1, 52–61.

T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. 2006. Optimizing memory transactions. In *Proc. of the 2006 Conference on Programming Language Design and Implementation* 41, 6, 14–25.

H. Kim, N. P. Johnson, J. W. Lee, S. A. Mahlke, and D. I. August. 2012. Automatic speculative doall for clusters. In *Proc. of the 2012 International Symposium on Code Generation and Optimization*. 94–103.

C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. 2008. DiSTM: A software transactional memory framework for clusters. In *Proc. of the 2008 International Conference on Parallel Processing*. 51–58.

M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Cascaval. 2009. How much parallelism is there in irregular applications? In *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 3–14.

M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. 2007. Optimistic parallelism requires abstractions. In *Proc. of the 2007 Conference on Programming Language Design and Implementation*. 211–222.

S. I. Lee, T. Johnson, and R. Eigenmann. 2003. Cetus—an extensible compiler infrastructure for source-to-source transformation. In *Proc. of the 16th Workshop on Languages and Compilers for Parallel Computing*.

V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. 2010. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on CPU and GPU. In *Proc. of the 37th Annual International Symposium on Computer Architecture*. 451–460.

A. Leung, O. Lhoták, and G. Lashari. 2009. Automatic parallelization for graphics processing units. In *Proc. of the 7th International Conference on Principles and Practice of Programming in Java*. 91–100.

S. Liu, C. Eisenbeis, and J.-L. Gaudiot. 2011. Value prediction and speculative execution on GPU. *International Journal of Parallel Programming* 39, 533–552.

M. Mehrara, J. Hao, P. chun Hsu, and S. Mahlke. 2009. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proc. of the 2009 Conference on Programming Language Design and Implementation*. 166–176.

J. Menon, M. de Kruijf, and K. Sankaralingam. 2012. iGPU: Exception support and speculative execution on GPUs. In *Proc. of the 39th Annual International Symposium on Computer Architecture*. 72–83.

NVIDIA. 2010. GPUs Are Only Up To 14 Times Faster than CPUs says Intel. Retrieved http://blogs.nvidia.com/ntersect/2010/06/gpus-are-only-up-to-14-times-faster-than-cpus-says-intel.html.

E. Nystrom, H.-S. Kim, and W. Hwu. 2004. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Proc. of the 11th Static Analysis Symposium*. 165–180.

C. E. Oancea and A. Mycroft. 2008. Software thread-level speculation: An optimistic library implementation. In *Proc. of the 1st International Workshop on Multicore Software Engineering*. 23–32.

Polybench. 2011. The Polyhedral Benchmark suite. Retrieved from http://www.cse.ohio-state.edu/pouchet/software/polybench.

L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. 2008. Iterative optimization in the polyhedral model: Part ii, multidimensional time. In *Proc. of the 2008 Conference on Programming Language Design and Implementation*. 90–100.

K. Psarris, X. Kong, and D. Klappholz. 1993. The direction vector I test. *IEEE Journal of Parallel Distributed Systems* 4, 11, 1280–1290.

L. Rauchwerger and D. A. Padua. 1999. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems* 10, 2, 160.

D. Roger, U. Assarsson, and N. Holzschuch. 2007. Efficient stream reduction on the GPU. In *Proc. of the 1st Workshop on General Purpose Processing on Graphics Processing Units*. 1–4.

S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W. mei W. Hwu. 2008. Program optimization space pruning for a multithreaded GPU. In *Proc. of the 2008 International Symposium on Code Generation and Optimization*. 195–204.

M. Samadi, A. Hormati, M. Mehrara, J. Lee, and S. Mahlke. 2012. Adaptive input-aware compilation for graphics engines. In *Proc. of the 2012 Conference on Programming Language Design and Implementation*. 13–22.

J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. 2005. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems* 23, 3, 253–300.

D. Tarditi, S. Puri, and J. Oglesby. 2006. Accelerator: Using data parallelism to program GPUs for general-purpose uses. In *Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. 325–335.

C. Tian, M. Feng, V. Nagarajan, and R. Gupta. 2008. Copy or discard execution model for speculative parallelization on multicores. In *Proc. of the 41st Annual International Symposium on Microarchitecture*. 330–341.

H. Volos, A. Welc, A.-R. Adl-Tabatabai, T. Shpeisman, X. Tian, and R. Narayanaswamy. 2009. NePalTM: Design and implementation of nested parallelism for transactional memory systems. In *Proc. of the 23nd European conference on Object-Oriented Programming*. 123–147.

R. P. Wilson, et al. 1994. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices* 29, 12, 31–37.

M. Wolfe. 1995. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman.

M. Wolfe. 2010. Implementing the PGI accelerator model. In *Proc. of the 3rd Workshop on General Purpose Processing on Graphics Processing Units*. 43–50.

S. Xiao and W. chun Feng. 2010. Inter-block GPU communication via fast barrier synchronization. In *Proc. of the 2010 IEEE International Symposium on Parallel and Distributed Processing*. 1–12.

E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. 2011. On-the-fly elimination of dynamic irregularities for GPU computing. In *Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. 369–380.

Received February 2013; revised July 2013; accepted October 2013